A User's Guide to the Z-Shell

Peter Stephenson

2003/03/23

Contents

Chapter 1

A short introduction

The Z-Shell, 'zsh' for short, is a command interpreter for UNIX systems, or in UNIX jargon, a 'shell', because it wraps around the commands you use. More than that, however, zsh is a particularly powerful shell — and it's free, and under regular maintenance — with lots of interactive features allowing you to do the maximum work with the minimum fuss. Of course, for that you need to know what the shell can do and how, and that's what this guide is for.

The most basic basics: I shall assume you have access to a UNIX system, otherwise the rest of this is not going to be much use. You can also use zsh under Windows by installing Cygwin, which provides a UNIX-like environment for programmes — given the weakness of the standard Windows command interpreter, this is a good thing to do. There are ports of older versions of zsh to Windows which run natively, i.e. without a UNIX environment, although these have a slightly different behaviour in some respects and I won't talk about them further.

I'll also assume some basic knowledge of UNIX; you should know how the filesystem works, i.e. what `/home/users/pws/.zshrc` and `../file` mean, and some basic commands, for example `ls`, and you should have experience with using `rm` to delete completely the wrong file by accident, and that sort of thing. In something like '`rm file`', I will often refer to the 'command' (`rm`, of course) and the 'argument(s)' (anything else coming after the command which is used by it), and to the complete thing you typed in one go as the 'command line'.

You're also going to need zsh itself; if you're reading this, you may well already have it, but if you don't, you or your system administrator should read Appendix A. For now, we'll suppose you're sitting in front of a terminal with zsh already running.

Now to the shell. After you log in, you probably see some prompt (a series of symbols on the screen indicating that you can input a command), such as '`$`' or '`%`', possibly with some other text in front — later, we'll see how you can change that text in interesting ways. That prompt comes from the shell. Type '`print hello`', then backspace over '`hello`' and type '`goodbye`'. Now hit the 'Return' key (or 'Enter' key, I'll just say `<RET>` from now on, likewise `<TAB>` for the tab key, `<SPC>` for the space key); unless you have a serious practical-joker problem on your system, you will see '`goodbye`', and the shell will come back with another prompt. All of the time up to when you hit `<RET>`, you were interacting with the shell and its editor, called 'Z-Shell Line Editor' or 'zle' for short; only then did the shell go away and tell the `print` command to print out a message. So you can see that the shell is important.

However, if all you're doing is typing simple commands like that, why do you need anything

complicated? In that case, you don't; but real life's not that simple. In the rest of this guide, I describe how, with zsh's help, you can:

- customise the environment in which you work, by using startup files,
- write your own commands to shorten tasks and store things in shell variables ('parameters') so you don't have to remember them,
- use zle to minimise the amount of typing you have to do — in zsh, you can even edit small files that way,
- pick the files you want to use for a particular command such as `mv` or `ls` using zsh's very sophisticated filename generation (known colloquially as 'globbing') system,
- tell the editor what sort of arguments you use with particular commands, so that you only need to type part of the name and it will complete the rest, using zsh's unrivalled programmable completion system,
- use the extra add-ons ('modules') supplied with the latest version of zsh to do other things you usually can't do in a shell at all.

That's only a tiny sample. Since there's so much to say, this guide will concentrate on the things zsh does best, and in particular the things it has which other shells don't. The next chapter gives a few of the basics, by trying to explain how to set the shell up the way you want it. Like the rest of the guide, it's not intended to be exhaustive, for which you should look at the shell manual.

Some other things you should probably know straight away. First, the shell is always running, even when the command you typed is running, too; the shell simply hangs around waiting for it to finish: you may know from other shells about putting commands in the **background** by putting an '`&`' after the command, which means that the shell doesn't wait for them to finish. The shell is there even if the command's in the foreground, but in this case doing nothing.

Second, it doesn't just run other people's commands, it has some of its own, called **builtin commands** or just **builtins**, and you can even add your own commands as lists of instructions to the shell called **functions**; builtins and functions always run in the shell itself. That's important to know, because things which don't run in the shell itself can't affect it, and hence can't alter parameters, functions, aliases, and all the other things I shall talk about.

1.1 Other shells and other guides

If you want a basic grounding in how shells work, what their syntax is (i.e. how to write commands), and how to write scripts and functions, you should read one of the many books on the subject. In particular, you will get most out of a book that describes the Korn shell (ksh), as zsh is very similar to this — so similar that it will be worth my while pointing out differences as we go along, since they can confuse ksh users. Recent versions of zsh can emulate ksh (strictly, the 1988 version of ksh, although there are increasingly features from the 1993 version) quite closely, although it's not perfect, and less perfect the more closely you look. However, it's important to realise that if you just start up any old zsh there is no guarantee that it will be set up to work like ksh; unless you or your system adminstrator have changed some settings, it certainly won't be. You might not see that straight away, but it affects the shell in subtle ways. I will talk about emulation a bit more later on.

A few other shells are worth mentioning. The grandfather of all UNIX shells is sh, now known as the Bourne shell but originally just referred to as 'the shell'. The story is similar to ksh: zsh can emulate sh quite closely (much more closely than ksh, since sh is considerably simpler), but in general you need to make sure it's set up to do that before you can be sure it will emulate sh.

You may also come across the 'Bourne-Again Shell', bash. This is a freely-available enhancement of sh written by the GNU project — but it is not always enhanced along the lines of ksh, and hence in many ways it is very different from zsh. On some free UNIX-like systems such as Linux/GNU (which is what people usually mean by Linux), the command sh is really bash, so there you should be extra careful when trying to ensure that something which runs under the so-called 'sh' will also run under zsh. Some Linux systems also have another simpler Bourne shell clone, ash; as it's simpler, it's more like the original Bourne shell.

Some more modern operating systems talk about 'the POSIX shell'. This is an attempt to standardize UNIX shells; it's most like the Korn shell, although, a bit confusingly, it's often just called sh, because the standard says that it should be. Usually, this just means you get a bit extra free with your sh and it still does what you expect. Zsh has made some attempts to fit the standard, but you have to tell it to — again, simply starting up 'zsh' will not have the right settings for that.

There is another common family of shells with, unfortunately, incompatible syntax. The source of this family is the C-Shell, csh, so called because its syntax looks more like the C programming language. This became widespread when the only other shell available was sh because csh had better interactive features, such as job control. It was then enhanced to make tcsh, which has many of the interactive features you will also find in zsh, and so became very popular. Despite these common features, the syntax of zsh is very different, so you should not try and use csh/tcsh commands beyond the very simplest in zsh; but if you are a tcsh user, you will find virtually every capability you are used to in zsh somewhere, plus a lot more.

1.2 Versions of zsh

At the time of writing, the most recent version of zsh available for widespread use was 4.0.6. You will commonly find two sets of older zsh's around. The 3.0 series, of which the last release was 3.0.9, was a stable release, with only bug fixes since the first release of zsh 3. The 3.1 series were beta versions, with lots of new features; the last of these, 3.1.9, was not so different from 4.0.1; the main change is that the shell has now been declared stable, so that as with zsh 3 there will be a set of bug fixes, labelled 4.0, and a set with new functions in, labelled 4.1. As 4.0 replaces all zsh 3 versions, I will try to keep things simple and talk about that; but every now and then it will be helpful to point out where older versions were different.

One notable feature of zsh is the completion of command line arguments. The system changed in 3.1.6 and 3.1.7 to make it a lot more configurable, and (provided you keep your wits about you) a little less obscure. I therefore won't describe the old completion system, which used the 'compctl' command, in any detail; a very brief introduction is given in the zsh FAQ. The old system remains available, however we strongly recommend new users to start with the new one. See chapter 6 'Completion, old and new' for the lowdown on new-style completion.

There won't be a big difference between 4.0 and 4.1, just bug fixes and a few evolutionary

changes, plus some extra modules. There will be some notes in chapter 7 about new features in 4.1, but nothing you write for 4.0 is likely to become obsolete in the foreseeable future.

1.3 Conventions

Most of what I say will be reasonably self-contained (which means I use phrases like 'as I said before' and 'as I'll discuss later on' more than a real stylist would like, and the number times I refer to other chapters is excessive), but there are some points I should perhaps draw your attention to before you leap in.

I will often write chunks of code as you would put them in a file for execution (a 'script' or a 'function', the differences to be discussed *passim*):

```
if [[ $ZSH_VERSION = 3.* ]]; then
  print This is a release of the third version of zsh.
else
  print This is either very new or very old.
fi
```

but sometimes I will show both what you type into a shell interactively, and what the shell throws back at you:

```
% print $ZSH_VERSION
3.1.9
% print $CPUTYPE
i586
```

Here, '`%`' shows the prompt the shell puts up to tell you it is expecting input (and the space immediately after is part of it). Actually, you probably see something before the percent sign like the name of the machine or your user name, or maybe something fancier. I've pruned it to the minimum to avoid confusion, and kept it as reminder that this is the line you type.

If you're reading an electronic version of this guide, and want to copy lines with the '`%`' in front into a terminal to be executed, there's a neat way of doing this where you don't even have to edit the line first:

```
alias %=' '
```

Then `%` at the start of a line is turned into nothing whatsoever; the space just indicates that any following aliases should be expanded. So the line '`% print $CPUTYPE`' will ignore the '`%`' and execute the rest of the line. (I hope it's obvious, but your *own* prompt is always ignored; this is just if you copy the prompts from the guide into the shell.)

There are lots of different types of object in zsh, but one of the most common is parameters, which I will always show with a '`$`' sign in front, like '`$ZSH_VERSION`', to remind you they are parameters. You need to remember that when you're setting or fiddling with the parameter itself, rather than its value, you omit the '`$`'. When you do and don't need it should become clearer as we go along.

The other objects I'll show specially are shell options — choices about how the shell is to work — which I write like this: '`SH_WORD_SPLIT`', '`NO_NOMATCH`', '`ZLE`'. Again, that's not the

whole story since whenever the shell expects options you can write them in upper or lower case with as many or as few underscores as you like; and often in code chunks I'll use the simplest form instead: '`shwordsplit`', '`nonomatch`', '`zle`'. If you're philosophical you can think of it as expressing the category difference between talking about programming and actual programming, but really it's just me being inconsistent.

1.4 Acknowledgments

I am grateful for comments from various zsh users. In particular, I have had detailed comments and corrections from Bart Schaefer, Sven 'Mr Completion' Wischnowsky and Oliver Kiddle. It's usual to add that any remaining errors are my own, but that's so stark staringly obvious as to be ridiculous. I mean, who wrote this? Never mind.

Most of this written on one or another release of Linux Mandrake (a derivative of Red Hat), with the usual GNU and XFree86 tools. Since all of this was free, it only seems fair to say 'thank you' for the gift. It also works a lot better than the operating system that came with this particular PC.

Chapter 2

What to put in your startup files

There are probably various changes you want to make to the shell's behaviour. All shells have 'startup' files, containing commands which are executed as soon as the shell starts. Like many others, zsh allows each user to have their own startup files. In this chapter, I discuss the sorts of things you might want to put there. This will serve as an introduction to what the shell does; by the end, you should have an inkling of many of the things which will be discussed in more detail later on and why they are interesting. Sometimes you will find out more than you want to know, such as how zsh differs from other shells you're not going to use. Explaining the differences here saves me having to lie about how the shell works and correcting it later on: most people will simply want to know how the shell normally works, and note that there are other ways of doing it.

2.1 Types of shell: interactive and login shells

First, you need to know what is meant by an **interactive** and a **login** shell. Basically, the shell is just there to take a list of commands and run them; it doesn't really care whether the commands are in a file, or typed in at the terminal. In the second case, when you are typing at a prompt and waiting for each command to run, the shell is **interactive**; in the other case, when the shell is reading commands from a file, it is, consequently, **non-interactive**. A list of commands used in this second way — typically by typing something like `zsh filename`, although there are shortcuts — is called a **script**, as if the shell was acting in a play when it read from it (and shells can be real hams when it comes to playacting). When you start up a script from the keyboard, there are actually two zsh's around: the interactive one you're typing at, which is waiting for another, non-interactive one to finish running the script. Almost nothing that happens in the second one affects the first; they are different copies of zsh.

Remember that when I give examples for you to type, I often show them as they would appear in a script, without prompts in front. What you actually see on the screen if you type them in will have a lot more in front.

When you first log into the computer, the shell you are presented with is interactive, but it is also a login shell. If you type '`zsh`', it starts up a new interactive shell: because you didn't give it the name of a file with commands in, it assumes you are going to type them interactively. Now you've got two interactive shells at once, one waiting for the other: it doesn't sound all that useful, but there are times when you are going to make some radical

changes to the shell's settings temporarily, and the easiest thing to do is to start another shell, do what you want to do, and exit back to the original, unaltered, shell — so it's not as stupid as it sounds.

However, that second shell will not be a login shell. How does zsh know the difference? Well, the programme that logs you in after you type your password (called, predictably, **login**), actually sticks a '-' in front of the name of the shell, which zsh recognises. The other way of making a shell a login shell is to run it yourself with the option `-l`; typing '`zsh -l`' will start a zsh that also thinks it's a login shell, and later I'll explain how to turn on options within the shell, which you can do with the login option too. Otherwise, any zsh you start yourself will not be a login shell. If you are using X-Windows, and have a terminal emulator such as xterm running a shell, that is probably not a login shell. However, it's actually possible to get xterm to start a login shell by giving it the option `-ls`, so if you type '`xterm -ls &`', you will get a window running a login shell (the `&` means the shell in the first window doesn't wait for it to finish).

The first main difference between a login shell and any other interactive shell is the one to do with startup files, described below. The other one is what you do when you're finished. With a login shell you can type '`logout`' to exit the shell; with another you type '`exit`'. However, '`exit`' works for all shells, interactive, non-interactive, login, whatever, so a lot of people just use that. In fact, the only difference is that '`logout`' will tell you '`not login shell`' if you use it anywhere else and fail to exit. The command '`bye`' is identical to '`exit`', only shorter and less standard. So my advice is just to use '`exit`'.

As somebody pointed out to me recently, login shells don't have to be interactive. You can always start a shell in the two ways that make it a login shell; the ways that make it an interactive shell or not are independent. In fact, some start-up scripts for windowing systems run a non-interactive login shell to incorporate definitions from the appropriate login scripts before executing the commands to start the windowing session.

2.1.1 What is a login shell? Simple tests

Telling if the shell you are looking at is interactive is usually easy: if there's a prompt, it's interactive. As you may have gathered, telling if it's a login shell is more involved because you don't always know how the shell was started or if the option got changed. If you want to know, you can type the following (one line at a time if you like, see below),

```
if [[ -o login ]]; then
  print yes
else
  print no
fi
```

which will print 'yes' or 'no' according to whether it's a login shell or not; the syntax will be explained as we go along. There are shorter ways of doing it, but this illustrates the commonest shell syntax for testing things, something you probably often want to do in a startup file. What you're testing goes inside the '`[[ ... ]]`'; in this case, the `-o` tells the shell to test an option, here `login`. The next line says what to do if the test succeeded; the line after the '`else`' what to do if the test failed. This syntax is virtually identical to ksh; in this guide, I will not give exhaustive details on the tests you can perform, since there are many of them, but just show some of the most useful. As always, see the manual — in this case, 'Conditional Expressions' in the `zshmisc` manual pages.

Although you usually know when a shell is interactive, in fact you can test that in exactly the same way, too: just use '`[[ -o interactive ]]`'. This is one option you can't change within the shell; if you turn off reading from the keyboard, where is the shell supposed to read from? But you can at least test it.

Aside for beginners in shell programming: maybe the semicolon looks a bit funny; that's because the '`then`' is really a separate command. The semicolon is just instead of putting it on a new line; the two are interchangeable. In fact, I could have written,

```
if [[ -o login ]]; then; print yes; else; print no; fi
```

which does exactly the same thing. I could even have missed out the semicolons after '`then`' and '`else`', because the shell knows that a command must come after each of those — though the semicolon or newline *before* the `then` is often important, because the shell does `not` know a command has to come next, and might mix up the `then` with the arguments of the command after the '`if`': it may look odd, but the '`[[ ... ]]`' is actually a command. So you will see various ways of dividing up the lines in shell programmes. You might also like to know that `print` is one of the builtin commands referred to before; in other words, the whole of that chunk of programme is executed by the shell itself. If you're using a newish version of the shell, you will notice that zsh tells you what it's waiting for, i.e. a '`then`' or an '`else`' clause — see the explanation of `$PS2` below for more on this. Finally, the spaces I put before the '`print`' commands were simply to make it look prettier; any number of spaces can appear before, after, or between commands and arguments, as long as there's at least one between ordinary words (the semicolon is recognised as special, so you don't need one before that, though it's harmless if you do put one in).

Second aside for users of sh: you may remember that tests in sh used a single pair of brackets, '`if [ ... ]; then ...`', or equivalently as a command called **test**, '`if test ...; then ...`'. The Korn shell was deliberately made to be different, and zsh follows that. The reason is that '`[[`' is treated specially, which allows the shell to do some extra checks and allows more natural syntax. For example, you may know that in sh it's dangerous to test a parameter which may be empty: '[$var = foo]' will fail if `$var` is empty, because in that case the word is missed out and the shell never knows it was supposed to be there; with '`[[ ... ]]`', this is quite safe because the shell is aware there's a word before the '=', even if it's empty. Also, you can use '`&&`' and '`||`' to mean logical 'and' and 'or', which agrees with the usual UNIX/C convention; in sh, they would have been taken as starting a new command, not as part of the test, and you have to use the less clear '`-a`' and '`-o`'. Actually, zsh provides the old form of test for backward compatibility, but things will work a lot more smoothly if you don't use it.

2.2 All the startup files

Now here's a list of the startup files and when they're run. You'll see they fall into two classes: those in the `/etc` directory, which are put there by the system administrator and are run for all users, and those in your home directory, which zsh, like many shells, allows you to abbreviate to a '~'. It's possible that the latter files are somewhere else; type '`print $ZDOTDIR`' and if you get something other than a blank line, or an error message telling you the parameter isn't set, it's telling you a directory other than '~' where your startup files live. If `$ZDOTDIR` (another parameter) is not already set, you won't want to set it without a good reason.

`/etc/zshenv` Always run for every zsh.

~/.zshenv Usually run for every zsh (see below).

/etc/zprofile Run for login shells.

~/.zprofile Run for login shells.

/etc/zshrc Run for interactive shells.

~/.zshrc Run for interactive shells.

/etc/zlogin Run for login shells.

~/.zlogin Run for login shells.

Now you know what login and interactive shells are, this should be straightforward. You may wonder why there are both `~/.zprofile` and `~/.zlogin`, when they are both for login shells: the answer is the obvious one, that one is run before, one after `~/.zshrc`. This is historical; Bourne-type shells run `/etc/profile`, and csh-type shells run `~/.login`, and zsh tries to cover the bases with its own startup files.

The complication is hinted at by the 'see below'. The file `/etc/zshenv`, as it says, is always run at the start of any zsh. However, if the option `NO_RCS` is set (or, equivalently, the `RCS` option is unset: I'll talk about options shortly, since they are important in startup files), none of the others are run. The most common way of setting this option is with a flag on the command line: if you start the shell as '`zsh -f`', the option becomes set, so only `/etc/zshenv` is run and the others are skipped. Often, scripts do this as a way of trying to get a basic shell with no frills, as I'll describe below; but if something is set in `/etc/zshenv`, there's no way to avoid it. This leads to the First Law of Zsh Administration: put as little as possible in the file `/etc/zshenv`, as every single zsh which starts up has to read it. In particular, if the script assumes that only the basic options are set and `/etc/zshenv` has altered them, it might well not work. So, at the absolute least, you should probably surround any option settings in `/etc/zshenv` with

```
if [[ ! -o norcs ]]; then
  ... <commands to run if NO_RCS is not set,
       such as setting options> ...
fi
```

and your users will be eternally grateful. Settings for interactive shells, such as prompts, have no business in `/etc/zshenv` unless you *really* insist that all users have them as defaults for every single shell. Script writers who want to get round problems with options being changed in `/etc/zshenv` should put '`emulate zsh`' at the top of the script.

There are two files run at the end: `~/.zlogout` and `/etc/zlogout`, in that order. As their names suggest, they are counterparts of the `zlogin` files, and therefore are only run for login shells — though you can trick the shell by setting the `login` option. Note that whether you use `exit`, `bye` or `logout` to leave the shell does not affect whether these files are run: I wasn't lying (this time) when I said that the error message was the only difference between `exit` and `logout`. If you want to run a file at the end of any other type of shell, you can do it another way:

```
TRAPEXIT() {
  # commands to run here, e.g. if you
  # always want to run .zlogout:
  if [[ ! -o login ]]; then
```

```
    # don't do this in a login shell
    # because it happens anyway
    . ~/.zlogout
  fi
}
```

If you put that in `.zshrc`, it will force `.zlogout` to be run at the end of all interactive shells. Traps will be mentioned later, but this is rather a one-off; it's really just a hack to get commands run at the end of the shell. I won't talk about logout files, however, since there's little that's standard to put in them; some people make them clear the screen to remove sensitive information with the '`clear`' command. Other than that, you might need to tidy a few files up when you exit.

2.3 Options

It's time to talk about options, since I've mentioned them several times. Each option describes one particular shell behaviour; they are all Boolean, i.e. can either be on or off, with no other state. They have short names and in the documentation and this guide they are written in uppercase with underscores separating the bits (except in actual code, where I'll write them in the short form). However, neither of those is necessary. In fact, `NO_RCS` and `norcs` and `__N_o_R_c_S__` mean the same thing and are all accepted by the shell.

The second thing is that an option with '`no`' in front just means the opposite of the option without. I could also have written the test '`[[ ! -o norcs ]]`' as '`[[ -o rcs ]]`'; the '`!`' means 'not', as in C. You can only have one '`no`'; '`nonorcs`' is meaningless. Unfortunately, there is an option '`NOMATCH`' which has '`no`' as part of its basic name, so in this case the opposite really is '`NO_NOMATCH`'; `NOTIFY`, of course, is also a full name in its own right.

The usual way to set and unset options is with the commands **setopt** and **unsetopt** which take a string of option names. Some options also have flags, like the '`-f`' for `NO_RCS`, which these commands also accept, but it's much clearer to use the full name and the extra time and space is negligible. The command '`set -o`' is equivalent to `setopt`; this comes from ksh. Note that `set` with no '`-o`' does something else — that sets the positional parameters, which is zsh's way of passing arguments to scripts and functions.

Almost everybody sets some options in their startup files. Since you want them in every interactive shell, at the least, the choice is between putting them in `~/.zshrc` or `~/.zshenv`. The choice really depends on how you use non-interactive shells. They can be started up in unexpected places. For example, if you use Emacs and run commands from inside it, such as **grep**, that will start a non-interactive shell, and may require some options. My rule of thumb is to put as many options as possible into `~/.zshrc`, and transfer them to `~/.zshenv` if I find I need them there. Some purists object to setting options in `~/.zshenv` at all, since it affects scripts; but, as I've already hinted, you have to work a bit harder to make sure scripts are unaffected by that sort of thing anyway. In the following, I just assume they are going to be in `~/.zshrc`.

2.4 Parameters

One more thing you'll need to know about in order to write startup files is parameters, also known as variables. These are mostly like variables in other programming languages.

Simple parameters can be stored like this (an **assignment**):

```
foo='This is a parameter.'
```

Note two things: first, there are no spaces around the '='. If there was a space before, zsh would think '`foo`' was the name of a command to execute; if there was a space after it, it would assign an empty string to the parameter `foo`. Second, note the use of quotes to stop the spaces inside the string having the same effect. Single quotes, as here, are the nuclear option of quotes: everything up to another single quote is treated as a simple string — newlines, equal signs, unprintable characters, the lot, in this example all would be assigned to the variable; for example,

```
foo='This is a parameter.
This is still the same parameter.'
```

So they're the best thing to use until you know what you're doing with double quotes, which have extra effects. Sometimes you don't need them, for example,

```
foo=oneword
```

because there's nothing in '`oneword`' to confuse the shell; but you could still put quotes there anyway.

Users of csh should note that you don't use '`set`' to set parameters. This is important because there is a `set` command, but it works differently — if you try '`set var="this wont't work"`', you won't get an error but you won't set the parameter, either. Type '`print $1`' to see what you did set instead.

To get back what was stored in a parameter, you use the name somewhere on the command line with a '`$`' tacked on the front — this is called an **expansion**, or to be more precise, since there are other types of expansion, a **parameter expansion**. For example, after the first assignment above.

```
print -- '$foo is "'$foo'"'
```

gives

```
$foo is "This is a parameter."
```

so you can see what I meant about the effect of single quotes. Note the asymmetry — there is no '`$`' when assigning the parameter, but there is a '`$`' in front to have it expanded it into the command line. You may find the word 'substitution' used instead of 'expansion' sometimes; I'll try and stick with the terminology in the manual.

Two more things while we're at it. First, why did I put '`--`' after the `print`? That's because **print**, like many UNIX commands, can take options after it which begin with a '-'. '`--`' says that there are no more options; so if what you're trying to print begins with a '-', it will still print out. Actually, in this case you can see it doesn't, so you're safe; but it's a good habit to get into, and I wish I had. As always in zsh, there are exceptions; for example, if you use the `-R` option to print before the '`--`', it only recognizes BSD-style options, which means it doesn't understand '`--`'. Indeed, zsh programmers can be quite lax about standards and often use

the old, but now non-standard, single '–' to show there are no more options. Currently, this works even after `-R`.

The next point is that I didn't put spaces between the single quotes and the `$foo` and it was still expanded — expansion happens anywhere the parameter is not quoted; it doesn't have to be on its own, just separated from anything which might make it look like a different parameter. This is one of those things that can help make shell scripts look so barbaric.

As well as defining your own parameters, there are also a number which the shell sets itself, and some others which have a special effect when you set them. All the above still applies, though. For the rest of this guide, I will indicate parameters with the '`$`' stuck in front, to remind you what they are, but you should remember that the '`$`' is missing when you set them, or, indeed, any time when you're referring to the name of the parameter instead of its value.

2.4.1 Arrays

There is a special type of parameter called an **array** which zsh inherited from both ksh and csh. This is a slightly shaky marriage, since some of the things those two shells do with them are not compatible, and zsh has elements of both, so you need to be careful if you've used arrays in either. The option `KSH_ARRAYS` is something you can set to make them behave more like they do in ksh, but a lot of zsh users write functions and scripts assuming it isn't set, so it can be dangerous.

Unlike normal parameters (known as **scalars**), arrays have more than one word in them. In the examples above, we made the parameter `$foo` get a string with spaces in, but the spaces weren't significant. If we'd done

```
foo=(This is a parameter.)
```

(note the absence of quotes), it would have created an array. Again, there must be no space between the '=' and the '(', though inside the parentheses spaces separate words just like they do on a command line. The difference isn't obvious if you try and print it — it looks just the same — but now try this:

```
print -- ${foo[4]}
```

and you get '`parameter.`'. The array stores the words separately, and you can retrieve them separately by putting the number of the element of the array in square brackets. Note also the braces '`{...}`' — zsh doesn't always require them, but they make things much clearer when things get complicated, and it's never wrong to put them in: you could have said '`${foo}`' when you wanted to print out the complete parameter, and it would be treated identically to '`$foo`'. The braces simply screen off the expansion from whatever else might be lying around to confuse the shell. It's useful too in expressions like '`${foo}s`' to keep the '`s`' from being part of the parameter name; and, finally, with `KSH_ARRAYS` set, the braces are compulsory, though unfortunately arrays are indexed from 0 in that case.

You can use quotes when defining arrays; as before, this protects against the shell thinking the spaces are between different elements of the array. Try:

```
foo=('first element' 'second element')
print -- ${foo[2]}
```

Arrays are useful when the shell needs to keep a whole series of different things together, so we'll meet some you may want to put in a startup file. Users of ksh will have noticed that things are a bit different in zsh, but for now I'll just assume you're using the normal zsh way of doing things.

2.5 What to put in your startup files

At the last count there were over 130 options and several dozen parameters which are special to the shell, and many of them deal with things I won't talk about till much later. But as a guide to get you started, and an indication of what's to come, here are some options and parameters you might want to think about setting in `~/.zshrc`.

2.5.1 Compatibility options: `SH_WORD_SPLIT` and others

I've already mentioned that zsh works differently from ksh, its nearest standard relative, and that some of these differences can be confusing to new users, for example the use of arrays. Some options like `KSH_ARRAYS` exist to allow you to have things work the ksh way. Most of these are fairly finnicky, but one catches out a lot of people. Above, I said that after

```
foo='This is a parameter.'
```

then `$foo` would be treated as one word. In traditional Bourne-like shells including sh, ksh and bash, however, the shell will split `$foo` on any spaces it finds. So if you run a command

```
command $foo
```

then in zsh the command gets a single argument '`This is a parameter.`', but in the other shells it gets the first argument '`This`', the second argument '`is`', and so on. If you like this, or are so used to it it would be confusing to change, you should set the option `SH_WORD_SPLIT` in your `~/.zshrc`. Most experienced zsh users use arrays when they want word splitting, since as I explained you have control over what is split and what is not; that's why `SH_WORD_SPLIT` is not set by default. Users of other shells just get used to putting things in double quotes,

```
command "$foo"
```

which, unlike single quotes, allow the '`$`' to remain special, and have the side effect that whatever is in quotes will remain a single word (though there's an exception to that, too: the parameter `$@`).

There are a lot of other options doing similar things to keep users of standard shells happy. Many of them simply turn features off, because the other shell doesn't have them and hence unexpected things might happen, or simply tweak a feature which is a little different or doesn't usually matter. Currently such options include `NO_BANG_HIST`, `BSD_ECHO` (sh only), `IGNORE_BRACES`, `INTERACTIVE_COMMENTS`, `KSH_OPTION_PRINT`, `NO_MULTIOS`, `POSIX_BUILTINS`, `PROMPT_BANG`, `SINGLE_LINE_ZLE` (I've written them how they would appear as an argument to `setopt` to put the option the way the other shell expects, so some have '`NO_`' in front). Most people probably won't change those unless they notice something isn't working how they expect.

Some others have more noticeable effects. Here are a few of the ones most likely to make you scratch your head if you're changing from another Bourne-like shell.

BARE_GLOB_QUAL, GLOB_SUBST, SH_FILE_EXPANSION, SH_GLOB, KSH_GLOB

These are all to do with how pattern matching works. You probably already know that the pattern '`*.c`' will be expanded into all the files in the current directory ending in '`.c`'. Simple uses like this are the same in all shells, and the way filenames are expanded is often referred to as 'globbing' for historical reasons (apparently it stood for 'global replacement'), hence the name of some of these options.

However, zsh and ksh differ over more complicated patterns. For example, to match either file `foo.c` or file `bar.c`, in ksh you would say `@(foo|bar).c`. The usual zsh way of doing things is `(foo|bar).c`. To turn on the ksh way of doing things, set the option `KSH_GLOB`; to turn off the zsh way, set the options `SH_GLOB` and `NO_BARE_GLOB_QUAL`. The last of those turns off **qualifiers**, a very powerful way of selecting files by type (for example, directories or executable files) instead of by name which I'll talk about in chapter 5.

The other two need a bit more explanation. Try this:

```
foo='*'
print $foo
```

In zsh, you usually get a '`*`' printed, while in ksh the '`*`' is expanded to all the files in the directory, just as if you had typed '`print *`'. This is a little like `SH_WORD_SPLIT`, in that ksh is pretending that the value of `$foo` appears on the command line just as if you typed it, while zsh is using what you assigned to `foo` without allowing it to be changed any more. To allow the word to be expanded in zsh, too, you can set the option `GLOB_SUBST`. As with `SH_WORD_SPLIT`, the way around the ksh behaviour if you don't want the value changed is to use double quotes: `"$foo"`.

You are less likely to have to worry about `SH_FILE_EXPANSION`. It determines when the shell expands things like `~/.zshrc` to the full path, e.g. `/home/user2/pws/.zshrc`. In the case of zsh, this is usually done quite late, after most other forms of expansion such as parameter expansion. That means if you set `GLOB_SUBST` and do

```
foo='~/.zshrc'
print $foo
```

you would normally see the full path, starting with a '/'. If you *also* set `SH_FILE_EXPANSION`, however, the '~' is tested much earlier, before `$foo` is replaced when there isn't one yet, so that '`~/.zshrc`' would be printed. This (with both options) is the way ksh works. It also means I lied when I said ksh treats `$foo` exactly as if its value had been typed, because if you type `print ~/.zshrc` the '~' does get expanded. So you see how convenient lying is.

NOMATCH, BAD_PATTERN

These also relate to patterns which produce file names, but in this case they determine what happens when the pattern doesn't match a file for some reason. There are two possible reasons: either no file happened to match, or you didn't use a proper pattern. In both cases, zsh, unlike ksh, prints an error message. For example,

```
% print nosuchfile*
zsh: no matches found: nosuchfile*
% print [-
zsh: bad pattern: [-
```

(Remember the '%' lines are what you type, with a prompt in front which comes from the shell.) You can see there are two different error messages: you can stop the first by setting NO_NOMATCH, and the second by setting NO_BAD_PATTERN. In both cases, that makes the shell print out what you originally type without any expansion when there are no matching files.

BG_NICE, NOTIFY

All UNIX shells allow you to start a *background* job by putting '&' at the end of the line; then the shell doesn't wait for the job to finish, so you can type something else. In zsh, such jobs are usually run at a lower priority (a 'higher nice value' in UNIX-speak), so that they don't use so much of the processor's time as foreground jobs (all the others, without the '&') do. This is so that jobs like editing or using the shell don't get slowed down, which can be highly annoying. You can turn this feature off by setting NO_BG_NICE.

When a background job finishes, zsh usually tells you immediately by printing a message, which interrupts whatever you're doing. You can stop this by setting NO_NOTIFY. Actually, this is an option in most versions of ksh, too, but it's a little less annoying in zsh because if it happens while you're typing something else to the shell, the shell will reprint the line you were on as far as you've got. For example:

```
% sleep 3 &
[1] 40366
% print The quick brown
[1]  + 40366 done       sleep 3
% print The quick brown
```

The command sleep simply does nothing for however many seconds you tell it, but here it did it in the background (zsh printed a message to tell you). After you typed for three seconds, the job exited, and with NOTIFY set it printed out another message: the 'done' is the key thing, as it tells you the job has finished. But zsh was smart enough to know the display was messed up, so it reprinted the line you were editing, and you can continue. If you were already running another programme in the foreground, however, that wouldn't know that zsh had printed the message, so the display would still be messed up.

HUP

Signals are the way of persuading a job to do something it doesn't want to, such as die; when you type ^C, it sends a signal (called SIGINT in this case) to the job. In zsh, if you have a background job running when the shell exits, the shell will assume you want that to be killed; in this case it is sent a particular signal called 'SIGHUP' which stands for 'hangup' (as in telephone, not as in Woody Allen) and is the UNIX equivalent of 'time to go home'. If you often start jobs that should go on even when the shell has exited, then you can set the option NO_HUP, and background jobs will be left alone.

KSH_ARRAYS

I've already mentioned this, but here are the details. Suppose you have defined an array `arr`, for example with

```
arr=(foo bar)
```

although the syntax in ksh, which zsh also allows, is

```
set -A arr foo bar
```

In zsh, `$arr` gives out the whole array; in ksh it just produces the first element. In zsh, `${arr[1]}` refers to the first element of the array, i.e. `foo`, while in ksh the first element is referred to as `${arr[0]}` so that `${arr[1]}` gives you `bar`. Finally, in zsh you can get away with `$arr[1]` to refer to an element, while ksh insists on the braces. By setting `KSH_ARRAYS`, zsh will switch to the ksh way of doing things. This is one option you need to be particularly careful about when writing functions and scripts.

FUNCTION_ARG_ZERO

Shell functions are a useful way of specifying a set of commands to be run by the shell. Here's a simple example:

```
% fn() { print My name is $0; }
% fn
My name is fn
```

Note the special syntax: the '`()`' appears after a function name to say you are defining one, then a set of commands appears between the '`{ ... }`'. When you type the name of the function, those commands are executed. If you know the programming language C, the syntax will be pretty familiar, although note that the '`()`' is a bit of a delusion: you might think you would put arguments to the function in there, but you can't, it must always appear simply as '`()`'. If you don't know C, it doesn't matter; nothing from C really applies in detail, it's just a superficial resemblance.

In this case, zsh printed the special parameter '`$0`' ('argument zero') and, as you see, that turned into the name of the function. Now `$0` outside a function means the name of the shell, or the name of the script for a non-interactive shell, so if you type '`print $0`' it will probably say '`zsh`'. In most versions of ksh, this is `$0`'s only use; it doesn't change in functions, and 'fn' would print 'ksh'. To get this behaviour, you can set `NO_FUNCTION_ARG_ZERO`. There's probably no reason why you would want to, but zsh functions quite often test their own name, so this is one reason why they might not work.

There's another difference when defining functions, irrespective of `FUNCTION_ARG_ZERO`: in zsh, you can get away without the final '`;`' before the end of the definition of `fn`, because it knows the '`}`' must finish the last command as well as the function; but ksh is not so forgiving here. Lots of syntactic know-alls will probably be able to tell you why that's a good thing, but fortunately I can't.

KSH_AUTOLOAD

There's an easy way of loading functions built into both ksh and zsh. Instead of putting them all together in a big startup file, you can put a single line in that,

```
autoload fn
```

and the function 'fn' will only be loaded when you run it by typing its name as a command. The shell needs to know where the function is stored. This is done by a special parameter called $fpath, an array which is a list of directories; it will search all the directories for a file called fn, and use that as the function definition. If you want to try this you can type 'autoload fn; fpath=(. $fpath)' and write a file called fn in the current directory.

Unfortunately ksh and zsh disagree a bit about what should be in that file. The normal zsh way of doing things is just putting the body of the function there. So if the file fn is autoloadable and contains,

```
# this is a simple function
print My name is $0
```

then typing 'fn' will have exactly the same effect as the function fn above, printing 'My name is fn'. Zsh users tend to like this because the function is written the same way as a script; if instead you had typed zsh fn, to call the file as a script with a new copy of zsh of its own, it would have worked the same way. The first line is a comment; it's ignored, and in zsh not even autoloaded when the function is run, so it's not only much clearer to add explanatory contents, it also doesn't use any more memory either. It uses more disk space, of course, but nowadays even home PCs come with the sort of disk size which allows you a little indulgence with legibility.

However, ksh does things differently, and here the file fn needs to contain

```
fn() {
  # this is a simple function
  print My name is $0
}
```

in other words, exactly what you would type to define the function. The advantage of this form is that you can put other things in the file, which will then be run straight away and forgotten about, such as defining things that fn may need to use but which don't need to be redefined every single time you run fn. The option to force zsh to work the ksh way here is called KSH_AUTOLOAD. (If you wanted to try the second example, you would need to type 'unfunction fn; autoload fn' to remove the function from memory and mark it for autoloading again.)

Actually, zsh is a little bit cleverer. If the option KSH_AUTOLOAD is not set, but the file contains just a function definition in the ksh form and nothing else (like the last one above, in fact), then zsh assumes that it needs to run the function just loaded straight away. The other possibility would be that you wanted to define a function which did nothing other than define a function of the same name, which is assumed to be unlikely — and if you really want to do that, you will need to trick zsh by putting a do-nothing command in the same file, such as a ':' on the last line.

A final complication — sorry, but this one actually happens — is that sometimes in zsh you want to define not just the function to be called, but some others to help it along. Then you need to do this:

```
fn() {
  # this is the function after which the file is named
}
helper() {
  # goodness knows what this does
}
fn "$@"
# this actually calls the function the first time,
# with any arguments passed (see the subsection
# 'Function Parameters' in the section 'Functions'
# of the next chapter for the "$@").
```

That last non-comment line is unnecessary with KSH_AUTOLOAD. The functions supplied with zsh assume that KSH_AUTOLOAD is not set, however, so you shouldn't turn it on unless you need to. You could just make fn into the whole body, as usual, and define helper inside that; the problem is that helper would be redefined each time you executed fn, which is inefficient. A better way of avoiding the problem would be to define helper as a completely separate function, itself autoloaded: in both zsh and ksh, it makes no difference whether a function is defined inside another function or outside it, unlike (say) Pascal or Scheme.

LOCAL_OPTIONS, LOCAL_TRAPS

These two options also refer to functions, and here the ksh way of doing things is usually preferable, so many people set at least LOCAL_OPTIONS in a lot of their functions. The first versions of zsh didn't have these, which is why you need to turn them on by hand.

If LOCAL_OPTIONS is set in a function (or was already set before the function, and not unset inside it), then any options which are changed inside the function will be put back the way they were when the function finishes. So

```
fn() {
  setopt localoptions kshglob
  ...
}
```

allows you to use a function with the ksh globbing syntax, but will make sure that the option KSH_GLOB is restored to whatever it was before when the function exits. This works even if the function was interrupted by typing ^C. Note that LOCAL_OPTIONS will itself be restored to the way it was.

The option LOCAL_TRAPS, which first appeared in version 3.1.6, is for a similar reason but refers to (guess what) **traps**, which are a way of stopping signals sent to the shell, for example by typing ^C to cancel something (SIGINT, short for 'signal interrupt'), or ^Z to suspend it temporarily (SIGTSTP, 'signal terminal stop'), or SIGHUP which we've already met, and so on. To do something of your own when the shell gets a ^C, you can do

```
trap 'print I caught a SIGINT' INT
```

and the set of commands in quotes will be run when the ^C arrives (you can even try it without running anything). If the string is empty (just '' with nothing inside), the signal will be ignored; typing ^C has no effect. To put it back to normal, the command is 'trap - INT'.

Traps are most useful in functions, where you may temporarily (say) not want things to stop when you hit ^C, or you may want to clear up something before returning from the function. So now you can guess what LOCAL_TRAPS does; with

```
fn() {
  setopt localoptions localtraps
  trap '' INT
  ...
}
```

the shell will ignore ^C's to the end of the function, but then put back the trap that was there before, or remove it completely if there was none. Traps are described in more detail in chapter 3.

There is a very convenient shorthand for making options and traps local, as well as for setting the others to their standard values: put 'emulate -L zsh' at the start of a function. This sets the option values back to the ones set when zsh starts, but with LOCAL_OPTIONS and LOCAL_TRAPS set too, so you now know exactly how things are going to work for the rest of the function, whatever options are set in the outside world. In fact, this only changes the options which affect normal programming; you can set every option which it makes sense to set to its standard value with 'emulate -RL zsh' (it doesn't, for example, make sense to change options like login at this point). Furthermore, you can make the shell behave as much like ksh as it knows how to by doing 'emulate -L ksh', with or without the -R.

The -L option to emulate actually only appears in versions from 3.0.6 and 3.1.6. Before that you needed

```
emulate zsh
setopt localoptions
```

since localtraps didn't exist, and indeed doesn't exist in 3.0.6 either.

PROMPT_PERCENT, PROMPT_SUBST

As promised, setting prompts will be discussed later, but for now there are two ways of getting information into prompts, such as the parameter $PS1 which determines the usual prompt at the start of a new command line. One is by using *percent escapes*, which means a '%' followed by another character, maybe with a number between the two. For example, the default zsh prompt is '%m%# '. The first percent escape turns into the name of the host computer, the second usually turns into a '%', but a '#' for the superuser. However, ksh doesn't have these, so you can turn them off by setting NO_PROMPT_PERCENT.

The usual ksh way of doing things, on the other hand, is by putting parameters in the prompt to be substituted. To get zsh to do this, you have to set PROMPT_SUBST. Then assigning

```
PS1='${PWD}% '
```

is another way of putting the name of the current directory ('$PWD' is presumably named after the command 'pwd' to 'print working directory') into the prompt. Note the single quotes, so that this happens when the prompt is shown, not when it is assigned. If they weren't there, or were double quotes, then the $PWD would be expanded to the directory when the assignment took place, probably your home directory, and wouldn't change to reflect the directory you were actually in. Of course, you need the quotes for the space, too, else it just gets swallowed up when the assignment is executed.

As there is potentially much more information available in parameters than the fixed number of predefined percent escapes, you may wish to set PROMPT_SUBST anyway. Furthermore, you can get the output of commands into prompts since other forms of expansion are done on them, not just that of parameters; in fact, prompts with PROMPT_SUBST are expanded pretty much the same as a string inside double quotes every time the prompt is displayed.

RM_STAR_SILENT

Everybody at some time or another deletes more files than they mean to (and *that's* a gross understatement); my favourite is:

```
rm *>o
```

That '>' should be a '.', but I still had the shift key pressed. This removes all files, echoing the output (there isn't any) into a file 'o'. Delightfully, the empty file 'o' is not removed. (Don't try this at home.)

There is a protection mechanism built into zsh to stop you deleting all the files in a directory by accident. If zsh finds that the command is 'rm', and there is a '*' on the command line (there may be other stuff as well), then it will ask you if you really want to delete all those files. You can turn this off by setting RM_STAR_SILENT. Overreliance on this option is a bad idea; it's only a last line of defence.

SH_OPTION_LETTERS

Many options also have single letters to stand for them; you can set an option in this way by, for example, 'set -f', which sets NO_RCS. However, even where sh, ksh and zsh share options, not all have the same letters. This option allows the single letter options to be more like those in sh and ksh. Look them up in the manual if you want to know, but I already recommended that you use the full names for options anyway.

SH_WORD_SPLIT

I've already talked about this, see above, but it's mentioned here so you don't forget it, since it's an important difference.

Starting zsh as ksh

Finally on the subject of compatibility, you might like to know that as well as 'emulate' there is another way of forcing zsh to behave as much like sh or ksh as possible. This is by actually

calling zsh under the name ksh. You don't need to rename zsh, you can make a link from the name zsh to the name ksh, which will be enough to convince it.

There is an easier way when you are doing this from within zsh itself. The parameter `$ARGV0` is special; it is the value which will be passed as the first argument of a command which is run by the shell. Normally this is the name of the command, but it doesn't have to be since the command only finds out what it is after it has already been run. You can use it to trick a programme into thinking its name is different. So

```
ARGV0=ksh zsh
```

will start a copy of zsh that tries to make itself like ksh. Note this doesn't work unless you're already in zsh, as the `$ARGV0` won't be special.

I haven't mentioned putting a parameter assignment before a command name, but that simply assigns the parameter (strictly an environment variable in this case) for the duration of the command; the value `$ARGV0` won't be set after that command (the ksh-like zsh) finishes, as you can easily test with `print`. While I'm here, I should mention a few of its other features. First, the parameter is automatically exported to the environment, meaning it's available for other programmes started by zsh (including, in this case, the new zsh) — see the section on environment variables below. Second, this doesn't do what you might expect:

```
FOO=bar print $FOO
```

because of the order of expansion: the command line and its parameters are expanded before execution, giving whatever value `$FOO` had before, probably none, then FOO=bar is put into the environment, and then the command is executed but doesn't use the new value of `$FOO`.

2.5.2 Options for csh junkies

As well as old ksh users, there are some options available to make old csh and tcsh users feel more at home. As you will already have noticed, the syntax is very different, so you are never going to feel completely at home and it might be best just to remember the fact. But here is a brief list. The last, `CSH_NULL_GLOB`, is actually quite useful.

CSH_JUNKIE_HISTORY

Zsh has the old csh mechanism for referring to words on a previous command line using a '`!`'; it's less used, now the editor is more powerful, but is still a convenient shorthand for extracting short bits from the previous line. This mechanism is sometimes called **bang-history**, since busy people sometimes like to say '`!`' as 'bang'. This option affects how a single '`!`' works. For example,

```
% print foo bar
% print open closed
% print !-2:1 !:2
```

In the last line, '`!-2`' means two entries ago, i.e. the line '`print foo bar`'. The '`:1`' chooses the first word after the command, i.e. '`foo`'. In the second expression, no number is given after the '`!`'. Usually zsh interprets that to mean that the same item just selected, in

this case -2, should be used. With CSH_JUNKIE_HISTORY set, it refers instead to the last command. Note that if you hadn't given that -2, it would refer to the last command in any case, although the explicit way of referring to the last command is '!!' — you have to use that if there are no ':' bits following. In summary, zsh usually gives you 'print foo bar'; with CSH_JUNKIE_HISTORY you get 'print foo closed'.

There's another option controlling this, BANG_HIST. If you unset that, the mechanism won't work at all. There's also a parameter, $histchars. The first character is the main history expansion character, normally '!' of course; the second is for rapid substitutions (normally '^' — use of this is described below); the third is the character introducing comments, normally '#'. Changing the third character is definitely not recommended. There's little real reason to change any.

CSH_JUNKIE_LOOPS

Normal zsh loops look something like this,

```
while true; do
  print Never-ending story
done
```

which just prints the message over and over (type it line-by-line at the prompt, if you like, then ^C to stop it). With CSH_JUNKIE_LOOPS set, you can instead do

```
while true
  print Never-ending story
end
```

which will, of course, make your zsh code unlike most other people's, so for most users it's best to learn the proper syntax.

CSH_NULL_GLOB

This is another of the family of options like NO_NOMATCH, already mentioned. In this case, if you have a command line consisting of a set of patterns, at least one of them must match at least one file, or an error is caused; any that don't match are removed from the command line. The default is that all of them have to match. There is one final member of this set of options, NULL_GLOB: all non-matching patterns are removed from the command line, no error is caused. As a summary, suppose you enter the command 'print file1* file2*' and the directory contains just the file file1.c.

1. By default, there must be files matching both patterns, so an error is reported.
2. With NO_NOMATCH set, any patterns which don't match are left alone, so 'file1.c file2*' is printed.
3. With CSH_NULL_GLOB set, file1* matched, so file2* is silently removed; 'file1.c' is reported. If that had not been there, an error would have been reported.
4. With NULL_GLOB set, any patterns which don't match are removed, so again 'file1.c' is printed, but in this case if that had not been there a blank line would have been printed, with no error.

CSH_NULL_GLOB is good thing to have set since it can keep you on the straight and narrow without too many unwanted error messages, so this time it's not just for csh junkies.

CSH_JUNKIE_QUOTES

Here just for completeness. Csh and friends don't allow multiline quotes, as zsh does; if you don't finish a pair of quotes before a new line, csh will complain. This option makes zsh do the same. But multi-line quotes are very useful and very common in zsh scripts and functions; this is only for people whose minds have been really screwed up by using csh.

2.5.3 The history mechanism: types of history

The name 'history mechanism' refers to the fact that zsh keeps a 'history' of the commands you have typed. There are three ways of getting these back; all these use the same set of command lines, but the mechanisms for getting at them are rather different. For some reason, items in the history list (a complete line of input typed and executed at once) have become known as 'events'.

Editing the history directly

First, you can use the editor; usually hitting up-arrow will take you to the previous line, and down-arrow takes you back. This is usually the easiest way, since you can see exactly what you're doing. I will say a great deal more about the editor in chapter 4; the first thing to know is that its basic commands work either like emacs, or like vi, so if you know one of those, you can start editing lines straight away. The shell tries to guess whether to use emacs or vi from the environment variables $VISUAL or $EDITOR, in that order; these traditionally hold the name of your preferred editor for programmes which need you to edit text. In the old days, $VISUAL was a full-screen editor and $EDITOR a line editor, like ed of blessed memory, but the distinction is now very blurred. If either contains the string vi, the line editor will start in vi mode, else it will start in emacs mode. If you're in the wrong mode, 'bindkey -e' in ~/.zshrc takes you to emacs mode and 'bindkey -v' to vi mode. For vi users, the thing to remember is that you start in insert mode, so type 'ESC' to be able to enter vi commands.

'Bang'-history

Second, you can use the csh-style 'bang-history' mechanism (unless you have set the option NO_BANG_HIST); the 'bang' is the exclamation mark, '!', also known as 'pling' or 'shriek' (or factorial, but that's another story). Thus '!!' retrieves the last command line and executes it; '!-2' retrieves the second last. You can select words: '!!:1' picks the first word after the command of the last command (if you were paying attention above, you will note you just need one '!' in that case); 0 after colon would pick the command word itself; '*' picks all arguments after the command; '$' picks the last word. You can even have ranges: '!!:1-3' picks those three words, and things like '!!:3-$' work too.

After the word selector, you can have a second set of colons and then some special commands called **modifiers** — these can be very useful to remember, since they can be applied to parameters and file patterns to, so here's some more details. The ':t' (tail) modifier picks the last part of a filename, everything after the last slash; conversely, ':h' (head) picks everything before that. So with a history entry,

```
% print /usr/bin/cat
/usr/bin/cat
% print !!:t
print cat
cat
```

Note two things: first, the bang-history mechanism always prints what it's about to execute. Secondly, you don't need the word selector; the shell can tell that the ':t' is a modifier, and assumes you want it applied to the entire previous command. (Be careful here, since actually the :t will reduce the expression to everything after the last slash in *any* word, which is a little unexpected.)

With parameters:

```
% foo=/usr/bin/cat
% print ${foo:h}
/usr/bin
```

(you can usually omit the '{' and '}', but it's clearer and safer with them). And finally with files — this won't work if you set NO_BARE_GLOB_QUAL for sh-like behaviour:

```
% print /usr/bin/cat(:t)
cat
```

where you need the parentheses to tell the shell the ':t' isn't just part of the file name.

For a complete list, see the zshexpn manual, or the section Modifiers in the printed or Info versions of the manual, but here are a few more of the most useful. ':r' removes the suffix of a file, turning file.c into file; ':l' and ':u' make the word(s) all lowercase or all uppercase; ':s/foo/bar/' substitutes the first occurrence of foo with bar in the word(s); ':gs/foo/bar' substitutes all occurrences (the 'g' stands for global); ':&' repeats the last such substitution, even if you did it on a previous line; ':g&' also works. So

```
% print this is this line
this is this line
% !!:s/this/that/
print that is this line
that is this line
% print this is no longer this line
this is no longer this line
% !!:g&
print that is no longer that line
that is no longer that line
```

Finally, there is a shortcut: ^old^new^ is exactly equivalent to !!:s/old/new/; you can even put another modifier after it. The '^' is actually the second character of $histchars mentioned above. You can miss out the last '^' if there's nothing else to follow it. By the way, you can put modifiers together, but each one needs the colon with it: :t:r applied to 'dir/file.c' produces 'file', and repeated applications of :h get you shorter and shorter paths.

Before we leave bang-history, note the option HIST_VERIFY. If that's set, then after a substitution the line appears again with the changes, instead of being immediately printed

and executed. As you just have to type <RET> to execute it, this is a useful trick to save you executing the wrong thing, which can easily happen with complicated bang-history lines; I have this set myself.

And one last tip: the shell's expansion and completion, which I will enthuse about at length later on, allows you to expand bang-history references straight away by hitting TAB immediately after you've typed the complete reference, and you can usually type control together with slash (on some keyboards, you are restricted to ^Xu) to put it back the way it was if you don't like the result — this is part of the editor's 'undo' feature.

Ksh-style history commands

The third form of history uses the fc builtin. It's the most cumbersome: you have to tell the command which complete lines to execute, and may be given a chance to edit them first (but using an external editor, not in the shell). You probably won't use it that way, but there are three things which are actually controlled by fc which you might use: first, the 'r' command repeats the last command (ignoring r's), which is a bit like '!!'. Secondly, the command called 'history' is also really fc in disguise. It gives you a list of recent commands. They have numbers next to them; you can use these with bang-history instead of using negative numbers to count backward in the way I originally explained, the advantage being they don't change as you enter more commands. You can give ranges of numbers to history, the first number for where to start listing, and the second where to stop: a particular example is 'history 1', which lists all commands (even if the first command it still remembers is higher than 1; it just silently omits all those). The third use of fc is for reading and writing your history so you can keep it between sessions.

2.5.4 Setting up history

In fact, the shell is able to read and write history without being told. You need to tell it where to save the history, however, and for that you have to set the parameter $HISTFILE to the name of the file you want to use (a common choice is '~/.history'). Next, you need to set the parameter $SAVEHIST to the number of lines of your history you want saved. When these two are set, the shell will read $HISTSIZE lines from $HISTFILE at the start of an interactive session, and save the last $SAVEHIST lines you executed at the end of the session. For it to read or write in the middle, you will either need to set one of the options described below (INC_APPEND_HISTORY and SHARE_HISTORY), or use the fc command: fc -R and fc -W read and write the history respectively, while fc -A appends it to the the file (although pruning it if it's longer than $SAVEHIST); fc -WI and fc -AI are similar, but the I means only write out events since the last time history was written.

There is a third parameter $HISTSIZE, which determines the number of lines the shell will keep within one session; except for special reasons which I won't talk about, you should set $SAVEHIST to be no more than $HISTSIZE, though it can be less. The default value for $HISTSIZE is 30, which is a bit stingy for the memory and disk space of today's computers; zsh users often use anything up to 1000. So a simple set of parameters to set in .zshrc is

```
HISTSIZE=1000
SAVEHIST=1000
HISTFILE=~/.history
```

and that is enough to get things working. Note that you *must* set $SAVEHIST and $HISTFILE

for automatic reading and writing of history lines to work.

2.5.5 History options

There are also many options affecting history; these increased substantially with version 3.1.6, which provided for the first time `INC_APPEND_HISTORY`, `SHARE_HISTORY`, `HIST_EXPIRE_DUPS_FIRST`, `HIST_IGNORE_ALL_DUPS`, `HIST_SAVE_NO_DUPS` and `HIST_NO_FUNCTIONS`. I have already described `BANG_HIST`, `CSH_JUNKIE_HISTORY` and `HIST_VERIFY` and I won't talk about them again.

APPEND_HISTORY, INC_APPEND_HISTORY, SHARE_HISTORY

Normally, when it writes a history file, zsh just overwrites everything that's there. `APPEND_HISTORY` allows it to append the new history to the old. The shell will make an effort not to write out lines which should be there already; this can get complicated if you have lots of zshs running in different windows at once. This option is a good one for most people to use. `INC_APPEND_HISTORY` means that instead of doing this when the shell exits, each line is added to the history in this way as it is executed; this means, for example, that if you start up a zsh inside the main shell its history will look like that of the main shell, which is quite useful. It also means the ordering of commands from different shells running at the same time is much more logical — basically just the order they were executed — so for 3.1.6 and higher this option is recommended.

`SHARE_HISTORY` takes this one stage further: as each line is added, the history file is checked to see if anything was written out by another shell, and if so it is included in the history of the current shell too. This means that zsh's running in different windows but on the same host (or more generally with the same home directory) share the same history. Note that zsh tries not to confuse you by having unexpected history entries pop up: if you use `!`-style history, the commands from other session don't appear in the history list until you explicitly type the `history` command to display them, so that you can be sure what command you are actually reexecuting. The Korn shell always behaves as if `SHARE_HISTORY` is set, presumably because it doesn't store history internally.

EXTENDED_HISTORY

This makes the format of the history entry more complicated: in addition to just the command, it saves the time when the command was started and how long it ran for. The `history` command takes three options which use this: `history -d` prints the start time of the command; `history -f` prints that as well as the date; `history -D` (which you can combine with `-f` or `-d`) prints the command's elapsed time. The date format can be changed with `-E` for European (*day.month.year*) and `-i` for international (*year-month-day*) formats. The main reasons why you *wouldn't* want to set this would be shortage of disk space, or because you wanted your history file to be read by another shell.

HIST_IGNORE_DUPS, HIST_IGNORE_ALL_DUPS, HIST_EXPIRE_DUPS_FIRST, HIST_SAVE_NO_DUPS, HIST_FIND_NO_DUPS

These options give ways of dealing with the duplicate lines that often appear in the history. The simplest is HIST_IGNORE_DUPS, which tells the shell not to store a history line if it's the same as the previous one, thus collapsing a lot of repeated commands down to one; this is a very good option to have set. It does nothing when duplicate lines are not adjacent, so for example alternating pairs of commands will always be stored. The next two options can help here: HIST_IGNORE_ALL_DUPS simply removes copies of lines still in the history list, keeping the newly added one, while HIST_EXPIRE_DUPS_FIRST is more subtle: it preferentially removes duplicates when the history fills up, but does nothing until then. HIST_SAVE_NO_DUPS means that whatever options are set for the current session, the shell is not to save duplicated lines more than once; and HIST_FIND_NO_DUPS means that even if duplicate lines have been saved, searches backwards with editor commands don't show them more than once.

HIST_ALLOW_CLOBBER, HIST_REDUCE_BLANKS

These allow the history mechanism to make changes to lines as they are entered. The first affects output redirections, where you use the symbol > to redirect the output of a command or set of commands to a named file, or use >> to append the output to that file. If you have the NO_CLOBBER option set, then

```
touch newfile
echo hello >newfile
```

fails, because the 'touch' command has created newfile and NO_CLOBBER won't let you overwrite (clobber) it in the next line. With HIST_ALLOW_CLOBBER, the second line appears in the history as

```
echo hello >|newfile
```

where the >| overrides NO_CLOBBER. So to get round the NO_CLOBBER you can just go back to the previous line and execute it without editing it.

The second option, HIST_REDUCE_BLANKS, will tidy up the line when it is entered into the history by removing any excess blanks that mean nothing to the shell. This can also mean that the line becomes a duplicate of a previous one even if it would not have been in its untidied form. It is smart enough not to remove blanks which are important, i.e. are quoted.

HIST_IGNORE_SPACE, HIST_NO_STORE, HIST_NO_FUNCTIONS

These three options allow you to say that certain lines shouldn't go into the history at all. HIST_IGNORE_SPACE means that lines which begin with a space don't go into the history; the idea is that you deliberately type a space, which is not otherwise significant to the shell, before entering any line you want to be forgotten immediately afterwards. In zsh 4.0.1 this is implemented so that you can always recall the immediately preceding line for editing, even if it had a space; but when the next line is executed and entered into the history, the line beginning with the space is forgotten.

HIST_NO_STORE tells the shell not to store history or fc commands. while HIST_NO_FUNCTIONS tells it not to store function definitions as these, though usually infrequent, can be tiresomely long. A function definition is anything beginning 'function funcname {...' or 'funcname () { ...'.

NO_HIST_BEEP

Finally, HIST_BEEP is used in the editor: if you try to scroll up or down beyond the end of the history list, the shell will beep. It is on by default, so use NO_HIST_BEEP to turn it off.

2.5.6 Prompts

Most people have some definitions in .zshrc for altering the prompt you see at the start of each line. I've already mentioned PROMPT_PERCENT (set by default) and PROMPT_SUBST (unset by default); I'll assume here you haven't changed these settings, and point out some of the possibilities with **prompt escapes**, sequences that start with a '%'. If you get really sophisticated, you might need to turn on PROMPT_SUBST.

The main prompt is in a parameter called either $PS1 or $PROMPT or $prompt; the reason for having all these names is historical — they come from different shells — so I'll just stick with the shortest. There is also $RPS1, which prints a prompt at the right of the screen. The point of this is that it automatically disappears if you type so far along the line that you run into it, so it can help make the best use of space for showing long things like directories.

$PS2 is shown when the shell is waiting for some more input, i.e. it knows that what you have typed so far isn't a complete line: it may contain the start of a quoted expression, but not the end, or the start of some syntactic structure which is not yet finished. Usually you will keep it different from $PS1, but all the same escapes are understood in all five prompts.

$PS3 is shown within a loop started by the shell's select mechanism, when the shell wants you to input a choice: see the zshmisc manual page as I won't say much about that.

$PS4 is useful in debugging: there is an option XTRACE which causes the shell to print out lines about to be executed, preceded by $PS4. Only from version 3.1.6 has it started to be substituted in the same way as the other prompts, though this turns out to be very useful — see 'Location in script or function' in the following list.

Here are some of the things you might want to include in your prompts. Note that you can try this out before you alter the prompt by using 'print -P': this expands strings just are as they are in prompts. You will probably need to put the string in single quotes.

The time

Zsh allows you lots of different ways of putting the time into your prompt with percent escapes. The simplest are %t and %T, the time in 12 and 24 hour formats, and %*, the same as %T but with seconds; you can also have the date as (e.g.) 'Wed 22' using %w, as '9/22/99' (US format) using %W, or as '99-09-22' (International format) using %D. However, there is another way of using %D to get many more possibilities: a following string in braces, '%D{...}' can contain a completely different set of percent escapes all of which refer to elements of the time and date. On most systems, the documentation for the strftime function will tell you what these are. zsh has a few of its own, given in the zshmisc manual

page in the PROMPT EXPANSION section. For example, I use %D{%L:%M} which gives the time in hours and minutes, with the hours as a single digit for 1 to 9; it looks more homely to my unsophisticated eyes.

You can have more fun by using the '%(*numX.true.false*)' syntax, where *X* is one of t or T. For t, if the time in minutes is the same as *num* (default zero), then *true* is used as the text for this section of the prompt, while *false* is used otherwise. T does the same for hours. Hence

```
PS1='%(t.Ding!.%D{%L:%M})%# '
```

prints the message 'Ding!' at zero minutes past the hour, and a more conventional time otherwise. The '%#' is the standard sequence which prints a '#' if you are the superuser (root), or a '%' for everyone else, which occurs in a lot of people's prompts. Likewise, you could use '%(30t.Dong!....' for a message at half past the hour.

The current directory

The sequence '%~' prints out the directory, with any home or named directories (see below) shortened to the form starting with ~; the sequence '%/' doesn't do that shortening, so usually '%~' is better. Directories can be long, and there are various ways to deal with it. First, if you are using a windowing system you can put the directory in the title bar, rather than anywhere inside the window. Second, you can use $RPS1 which disappears when you type near it. Third, you can pick segments out of '%~' or '%/' by giving them a number after the '%': for example, '%1~' just picks out the last segment of the path to the current directory.

The fourth way gives you the most control. Prompts or parts of prompts, not just bits showing the directory, can be truncated to any length you choose. To truncate a path on the left, use something like '%10<...<%~'. That works like this: the '%<<' is the basic form for truncation. The 10 after the '%' says that anything following is limited to 10 characters, and the characters '...' are to be displayed whenever the prompt would otherwise be longer than that (you can leave this empty). This applies to anything following, so now the %~ can't be longer than 10 characters, otherwise it will be truncated (to 7 characters, once the '. . .' has been printed). You can turn off truncation with '%<<', i.e. no number after the '%'; truncation then applies to the entire region between where it was turned on and where it was turned off (this has changed from older versions of zsh, where it just applied to individual '%' constructs).

What are you waiting for?

The prompt $PS2 appears when the shell is waiting for you to finish entering something, and it's useful to know what the shell is waiting for. The sequence '%_' shows this. It's part of the default $PS2, which is '%_> '. Hence, if you type 'if true; then' and <RET>, the prompt will say 'then> '. You can also use it in the trace prompt, $PS4, to show the same information about what is being executed in a script or function, though as there is usually enough information there (as described next) it's not part of the default. In this case, a number after the '%' will limit the depth shown, so with '%1_' only the most recent thing will be mentioned.

Location in script or function

The default $PS4 contains '%N' and '%i', which tell you the name of the most recently started function, script, or sourced file, and the line number being executed inside it; they are not very useful in other prompts. However, '%i' in $PS1 will tell you the current interactive line number, which zsh keeps track of, though doesn't usually show you; the parameter $LINENO contains the same information.

Another point to bear about '%i' in mind is that the line number shown applies to the version of a function first read in, not how it appears with the 'functions' command, which is tidied up. If you use autoloaded functions, however, the file containing the function will usually be what you want to alter, so this shouldn't be a problem when debugging.

Remember, the $PS4 display only happens when the XTRACE option is set; as options may be local to functions, and always are to scripts, you will often need to put an explicit 'setopt xtrace' at the top of whatever you are debugging. Alternatively, you can use 'typeset -ft *funcname*' to turn on tracing for that function (something I only just discovered); use 'typeset +ft *funcname*' to turn it off again.

Other bits and pieces

There are many other percent escapes described in the zshmisc manual page, mostly straightforward. For example, '%h' shows you the history entry number, useful if you are using bang-history; '%m' shows you the current host name up to any dot; '%n' shows the username.

There are two other features I happen to use myself. First, it's sometimes convenient to know when the last command failed. Every command returns a status, which is a number: zero for success, some other number for some type of failure. You can get this from the parameter '$?' or '$status' (again, they refer to the same thing). It's also available in the prompt as '%?', and there's also one of the so-called 'ternary' expressions with parentheses I described for time, which pick different strings depending on a test. Here the test is, reasonably enough, '%(?...'. Putting these two together, you can get a message which is only displayed when the exit status is non-zero; I've put an extra set of parentheses around the number just to make it clearer, where the ')' needs to be turned into '%)' to stop it marking the end of the group:

```
PS1='%(?..(%?%))%# '
```

It's also sometimes convenient to know if you're in a subshell, that is if you've started another shell within the main one by typing 'zsh'. You can do this by using another ternary expression:

```
PS1='%(2L.+.)%# '
```

This checks the parameter SHLVL, which is incremented every time a new zsh starts, so if there was already one running (which would have set SHLVL to 1), it will now be 2; and if SHLVL is at least 2, an extra '+' is printed in front of the prompt, otherwise nothing. If you're using a windowing system, you may need to turn the 2 into 3 as there may be a zsh already running when you first log in, so that the shells in the windows have SHLVL set to 2 already. This depends a good deal on how your windowing system is set up; finding out more is left as an exercise for the reader.

Colours

Many terminals can now display colours, and it is quite useful to be able to put these into prompts to distinguish those from the surrounding text. I often find a programme has just dumped a whole load of output on my terminal and it's not obvious where it starts. Being able to find the prompt just before helps a lot.

Colors, like bold or underlined text, use escape sequences which don't move the cursor. The golden rule for inserting any such escape sequences into prompts is to surround them with '`%{`' at the start and '`%}`' at the end. Otherwise, the shell will be confused about the length of the line. This affects what happens when the line editor needs to redraw the line, and also changes the position of the right prompt `$RPS1`, if you use that. You don't need that with the special sequences `%B` and `%b`, which start and stop bold text, because the shell already knows what to do with those; it's only random characters which you happen to know don't move the cursor, though the shell doesn't, that cause the problem.

In the case of colours, there is a shell function `colors` supplied with the standard distribution to help you. When loaded and run, it defines associative array parameters `$fg` and `$bg` which you use to extract the escape sequences for given colours, for example `${fg[red]}${bg[yellow]}` produces the sequences for red text on a yellow background. So for example,

```
PS1="%{${bg[white]}${fg[red]}%}%(?..(%?%))\
%{${fg[yellow]}${bg[black]}%}%# "
```

produces a red-on-white '`(1)`' if the previous programme exited with status 1, but nothing if it exited with status 0, followed by a yellow-on-black '`%`' or '`#`' if you are the superuser. Note the use of the double quotes here to force the parameters to be expanded straight away — the escape sequences are fixed, so they don't need to be re-extracted from the parameters every time the prompt is shown.

Even if your terminal does support colour, there's no guarantee all the possibilities work, although the basic ANSI colour scheme is fairly standard. The colours understood are: cyan, white, yellow, magenta, black, blue, red, grey, green. You can also used 'default', which puts the terminal back how it was to begin with. In addition, you can use the basic colours with the parameters `$bg_bold` and `$fg_bold` for bold varieties of the colours and `$bg_no_bold` and `$fg_no_bold` to switch explicitly back to non-bold.

Themes

There are also a set of themes provided as functions to set up your prompt to various predefined possibilities. These make use of the colours set up as described above. See the `zshcontrib` manual page for how to do this (search for 'prompt themes').

2.5.7 Named directories

As already mentioned, '`~/`' at the start of a filename expands to your home directory. More generally, '*~user/*' allows you to refer to the home directory of any other user. Furthermore, zsh lets you define your own named directories which use this syntax. The basic idea is simple, since any parameter can be a named directory:

```
dir=/tmp/mydir
print ~dir
```

prints '/tmp/mydir'. So far, this isn't any different from using the parameter as `$dir`. The difference comes if you use the '`%~`' construct, described above, in your prompt. Then when you change into that directory, instead of seeing the message '`/tmp/mydir`', you will see the abbreviation '`~dir`'.

The shell will not register the name of the directory until you force it to by using '`~dir`' yourself at least once. You can do the following in your `.zshrc`:

```
dir=/tmp/mydir
bin=~/myprogs/bin
: ~dir ~bin
```

where '`:`' is a command that does nothing — but its arguments are checked for parameters and so on in the usual way, so that the shell can put `dir` and `bin` into its list of named directories. A more simple way of doing this is to set the option `AUTO_NAME_DIRS`; then any parameter created which refers to a directory will automatically be turned into a name. The directory must have an absolute path, i.e. its expanded value, after turning any '~'s at the start into full paths, must begin with a '/'. The parameter `$PWD`, which shows the current directory, is protected from being turned into `~PWD`, since that would tell you nothing.

2.5.8 'Go faster' options for power users

Here are a few more random options you might want to set in your `.zshrc`.

NO_BEEP

Normally zsh will beep if it doesn't like something. This can get extremely annoying; '`setopt nobeep`' will turn it off. I refer to this informally as the `OPEN_PLAN_OFFICE_NO_VIGILANTE_ATTACKS` option.

AUTO_CD

If this option is set, and you type something with no arguments which isn't a command, zsh will check to see if it's actually a directory. If it is, the shell will change to that directory. So '`./bin`' on its own is equivalent to '`cd ./bin`', as long as the directory '`./bin`' really exists. This is particularly useful in the form '`..`', which changes to the parent directory.

CD_ABLE_VARS

This is another way of saving typing when changing directory, though only one character. If a directory doesn't exist when you try to change to it, zsh will try and find a parameter of that name and use that instead. You can also have a '/' and other bits after the parameter. So 'cd `foo/dir`', if there is no directory '`foo`' but there is a parameter `$foo`, becomes equivalent to 'cd `$foo/dir`'.

EXTENDED_GLOB

Patterns, to match the name of files and other things, can be very sophisticated in zsh, but to get the most out of them you need to use this option, as otherwise certain features are not enabled, so that people used to simpler patterns (maybe just '`*`', '`?`' and '`[...]`') are not confused by strange happenings. I'll say much more about zsh's pattern features, but this is to remind you that you need this option if you're doing anything clever with '`~`', '`#`', '`^`' or globbing flags — and also to remind you that those characters can have strange effects if you have the option set.

MULTIOS

I mentioned above that to get zsh to behave like ksh you needed to set `NO_MULTIOS`, but I didn't say what the `MULTIOS` option did. It has two different effects for output and input.

First, for output. Here it's an alternative to the `tee` programme. I've mentioned once, but haven't described in detail, that you could use `>filename` to tell the shell to send output into a file with a given name instead of to the terminal. With `MULTIOS` set, you can have more than one of those redirections on the command line:

```
echo foo >file1 >file2
```

Here, '`foo`' will be written to **both** the named files; zsh copies the output. The pipe mechanism, which I'll describe better in chapter 3, is a sort of redirection into another programme instead of into a file: `MULTIOS` affects this as well:

```
echo foo >file1 | sed 's/foo/bar/'
```

Here, '`foo`' is again written to `file1`, but is also sent into the pipe to the programme `sed` ('stream editor') which substitutes '`foo`' into '`bar`' and (since there is no output redirection in this part) prints it to the terminal.

Note that the second example above has several times been reported as a bug, often in a form like:

```
some_command 2>&1 >/dev/null | sed 's/foo/bar/'
```

The intention here is presumably to send standard error to standard output (the '`2>&1`', a very commonly used shell hieroglyphic), and not send standard output anywhere (the '`>/dev/null`'). (If you haven't met the concept of 'standard error', it's just another output channel which goes to the same place as normal output unless you redirect it; it's used, for example to send error messages to the terminal even if your output is going somewhere else.) In this example, too, the `MULTIOS` feature forces the original standard output to go to the pipe. You can see this happening if we put in a version of '`some_command`':

```
{ echo foo error >&2; echo foo not error;  } 2>&1 >/dev/null |
 sed 's/foo/bar/'
```

where you can consider the stuff inside the '`{...}`' as a black box that sends the message 'foo error' to standard error, and 'foo not error' to standard output. With `MULTIOS`, however, the result is

```
error bar
 not error bar
```

because both have been sent into the pipe. Without MULTIOS you get the expected result,

```
error bar
```

as any other Bourne-style shell would produce. There

On input, MULTIOS arranges for a series of files to be read in order. This time it's a bit like using the programme cat, which combines all the files listed after it. In other words,

```
cat file1 file2 | myprog
```

(where myprog is some programme that reads all the files sent to it as input) can be replaced by

```
myprog <file1 <file2
```

which does the same thing. Once again, a pipe counts as a redirection, and the pipe is read from first, before any files listed after a '<':

```
echo then this >testfile
echo this first | cat <testfile
```

CORRECT, CORRECT_ALL

If you have CORRECT set, the shell will check all the commands you type and if they don't exist, but there is one with a similar name, it will ask you if you meant that one instead. You can type 'n' for no, don't correct, just go ahead; 'y' for yes, correct it then go ahead; 'a' for abort, don't do anything; 'e' for edit, return to the editor to edit the same line again. Users of the new completion system should note this is not the same correction you get there: it's just simple correction of commands.

CORRECT_ALL applies to all the words on the line. It's a little less useful, because currently the shell has to assume that they are supposed to be filenames, and will try to correct them if they don't exist as such, but of course many of the arguments to a command are not filenames. If particular commands generate too many attempts to correct their arguments, you can turn this off by putting 'nocorrect' in front of the command name. An alias is a very good way of doing this, as described next.

2.5.9 aliases

An alias is used like a command, but it expands into some other text which is itself used as a command. For example,

```
alias foo='print I said foo'
foo
```

prints (guess what) '`I said foo`'. Note the syntax for definition — you need the '=', and you need to make sure the whole alias is treated by the shell as one word; you can give a whole list of aliases to the same '`alias`' command. You may be able to think of some aliases you want to define in your startup files; `.zshrc` is probably the right place. If you have `CORRECT_ALL` set, the way to avoid the '`mkdir`' command spell-checking its arguments — which is useless, because they *have* to be non-existent for the command to work — is to define:

```
alias mkdir='nocorrect mkdir'
```

This shows one useful feature about aliases: the alias can contain something of the same name as itself. When it is encountered in the expansion text (the right hand side), the shell knows it is not to expand the alias again, but this time to treat it as a real command. Note that functions do *not* have this property: functions are more powerful than aliases and in some cases it is useful for them to call themselves, It's a common mistake to have functions call themselves over and over again until the shell complains. I'll describe ways round this in chapter 3.

One other way functions are more powerful than aliases is that functions can take arguments while aliases can't — in other words, there is no way of referring inside the alias to what follows it on the command line, unlike a function, and also unlike aliases in csh (because that has no functions, that's why). It is just blindly expanded, and the remainder of the command line stuck on the end. Hence aliases in zsh are usually kept for quite simple things, and functions are written for anything more complicated. You couldn't do that trick with '`nocorrect`' using a function, though, since the function is called too late: aliases are expanded straight away, so the `nocorrect` is found in time to be useful. You can almost think of them as just plain typing abbreviations.

Normal aliases only work when in command position, i.e. at the start of the command line (more strictly, when zsh is expecting a command). There are other things called 'global aliases', which you define by the '`-g`' option to `alias`, which will be expanded at any position on the command line. You should think seriously before defining these, as they can have a drastic effect. Note, however, that quoting a word, or even a single character, will stop an alias being expanded for it.

I only tend to use aliases in interactive shells, so I define them from `.zshrc`, but you may want to use `.zshenv` if you use aliases more widely. In fact, to keep my `.zshrc` neat I save all the aliases in a separate file called `.aliasrc` and in `.zshrc` I have:

```
if [[ -r ~/.aliasrc ]]; then
  . ~/.aliasrc
fi
```

which checks if there is a readable file `~/.aliasrc`, and if there is, it runs it in exactly the same way the normal startup files are run. You can use '`source`' instead of '`.`' if it means more to you; '`.`' is the traditional Bourne and Korn shell name, however.

2.5.10 Environment variables

Often, the manual for a programme will tell you to define certain environment variables, usually a collection of uppercase letters with maybe numbers and the odd underscore. These can pass information to the programme without you needing to use extra arguments. In zsh, environment variables appear as ordinary shell parameters, although they have to be

defined slightly differently: strictly, the environment is a special region outside the shell, and zsh has to be told to put a copy there as well as keeping one of its own. The usual syntax is

```
export VARNAME='value'
```

in other words, like an ordinary assignment, but with '`export`' in front. Note there is no '`$`' before the name of the environment variable; all '`export`' and similar statements work the same way. The easiest place to put these is in `.zshenv` — hence it's name. Environment variables will be passed to any programmes run from a shell, so it may be enough to define them in `.zlogin` or `.zprofile`: however, any shell started for you non-interactively won't run those, and there are other possible problems if you use a windowing system which is started by a shell other than zsh or which doesn't run a shell start-up file at all — I had to tweak mine to make it do so. So `.zshenv` is the safest place; it doesn't take long to define environment variables. Other people will no doubt give you completely contradictory views, but that's people for you.

Note that you can't export arrays. If you export a parameter, then assign an array to it, nothing will appear in the environment; you can use the external command '`printenv VARNAME`' (again no '`$`' because the command needs to know the name, not the value) to check. There's a more subtle problem with arrays, too. The `export` builtin is just a special case of the builtin **typeset**, which defines a variable without marking it for export to the environment. You might think you could do

```
typeset array=(this doesn\'t work)
```

but you can't — the special array syntax is only understood when the assignment does not follow a command, not in normal arguments like the case here, so you have to put the array assignment on the next line. This is a very easy mistake to make. More uses of `typeset` will be described in chapter 3; they include creating local parameters in functions, and defining special attributes (of which the 'export' attribute is just one) for parameters.

2.5.11 Path

It helps to be able to find external programmes, i.e. anything not part of the shell, any command other than a builtin, function or alias. The `$path` array is used for this. Actually, what the system needs is the environment variable `$PATH`, which contains a list of directories in which to search for programmes, separated from each other by a colon. These directories are the individual components of the array `$path`. So if `$path` contains

```
path=(/bin /usr/bin /usr/local/bin .)
```

then `$PATH` will automatically contain the effect of

```
PATH=/bin:/usr/bin:/usr/local/bin:.
```

without you having to set that. The idea is simply that, while the system needs `$PATH` because it doesn't understand arrays, it's much more flexible to be able to use arrays within the shell and hence pretty much forget about the `$PATH` form.

Changes to the path are similar to changes to environment variables described above, so all that applies. There's a slight difficulty in setting `$path` in `.zshenv` however, even though

the reasons given above for doing so still apply. Usually, the path will be set for you, either by the system, or by the system administrator in one of the global start up files, and if you change path you will simply want to add to it. But if your `.zshenv` contains

```
path=(~/bin ~/progs/bin $path)
```

— which is the right way of adding something to the front of `$path` — then every time `.zshenv` is called, `~/bin` and `~/progs/bin` are stuck in front, so if you start another zsh you will have two sets there.

You can add tests to see if something's already there, of course. Zsh conveniently allows you to test for the existence of elements in an array. By preceding an array index by `(r)` (for reverse), it will try to find a matching element and return that, else an empty string. Here's a way of doing that (but don't add this yet, see the next paragraph):

```
for dir in ~/bin ~/progs/bin; do
  if [[ -z ${path[(r)$dir]} ]]; then
    path=($dir $path)
  fi
done
```

That `for... do ... done` is another special shell construct. It takes each thing after '`in`' and assigns it in turn to the parameter named before the '`in`' — `$dir`, but because this is a form of assignment, the '`$`' is left off — so the first time round it has the effect of `dir=~/bin`, and the next time `dir=~/progs/bin`. Then it executes what's in the loop. The test `-z` checks that what follows is empty: in this case it will be if the directory `$dir` is not yet in `$path`, so it goes ahead and adds it in front. Note that the directories get added in the reverse of the order they appear.

Actually, however, zsh takes all that trouble away from you. The incantation '`typeset -U path`', where the `-U` stands for unique, tells the shell that it should not add anything to `$path` if it's there already. To be precise, it keeps only the left-most occurrence, so if you added something at the end it will disappear and if you added something at the beginning, the old one will disappear. Thus the following works nicely in `.zshenv`:

```
typeset -U path
path=(~/bin ~/progs/bin $path)
```

and you can put down that '`for`' stuff as a lesson in shell programming. You can list all the variables which have uniqueness turned on by typing '`typeset +U`', with '`+`' instead of '`-`', because in the latter case the shell would show the values of the parameters as well, which isn't what you need here. The `-U` flag will also work with colon-separated arrays, like `$PATH`.

2.5.12 Mail

Zsh will check for new mail for you. If all you need is to be reminded of something arriving in your normal folder every now and then, you just need to set the parameter `$MAIL` to wherever that is: it's typically one of `/usr/spool/mail`, `/var/spool/mail`, or `/var/mail`.

The array `$mailpath` allows more possibilities. Like `$path`, it has a colleague in uppercase, `$MAILPATH`, which is a colon-separated array. The system doesn't need that, this time, so

it's mainly there so that you can export it to another version of zsh; exporting arrays won't work. As may by now be painfully clear, if you set in `.zshenv` or `.zshrc`, you don't need to export it, because it's set in each instance of the shell. The elements of `$mailpath` work like `$MAIL`, so you can specify different places where mail arrives. That's most useful if you have a programme like `filter` or `procmail` running to redistribute arriving mail to different folders. You can specify a different message for each folder by putting '?*message*' at the end. For example, mine looks like this.

```
mailpref=/temp/pws/Mail
mailpath=($mailpref/newmail
          $mailpref/zsh-new'?New zsh mail'
          $mailpref/list-new'?New list mail'
          $mailpref/urth-new'?New Urth mail')
```

Note that zsh knows the array isn't finished until the ')', even though the elements are on different lines; this is one very good reason for setting `$mailpath` rather than `$MAILPATH`, which needs one long chunk.

The other parameter of interest is `$MAILCHECK`, which gives the frequency in seconds when zsh should check for new mail. The default is 60. Actually, zsh only checks just after a command has finished running and it is about to print a prompt. Since checking files doesn't take long, you can usually set this to its minimum value, which is `MAILCHECK=1`; zero doesn't work because it switches off checking. One reason why you wouldn't want to do that might be because `$MAIL` and `$mailpath` can contain directories instead of ordinary files; these will be checked recursively for any files with something new in them, so this can be slow.

Finally, there is one associated option, `MAIL_WARNING` (though `MAIL_WARN` is also accepted for the same thing for reasons of compatibility with less grammatical shells). The shell remembers when it found the mail file was checked; next time it checks, it compares the date. If there is no new mail, but the date of the file changed anyway, it will print a warning message. This will happen if you read the mail with your mail reader and put the messages somewhere else. Presumably you *know* you did that, so the warning may not be all that useful.

2.5.13 Other path-like things

There are other pairs like `$path` and `$PATH`. I will keep back talk of `$cdpath` until I say more about the way zsh handles directories. When I mentioned `$fpath`, I didn't say there was also `$FPATH`, but there is. Then there is `$manpath` and `$MANPATH`; these aren't used by the shell at all, but `$MANPATH`, if exported, is used by the **man** external command, and `$manpath` gives an easier way to set it.

From 3.1.6 there is a mechanism to define your own such combinations; if this had been available before, there would have been no need to build in `$manpath` and `$MANPATH`. In `.zshenv` you would put,

```
export -TU TEXINPUTS texinputs
```

to define such a pair. The `-T` (for tie) is the key to that; I've used '`export`' even though the basic variable declaration command is '`typeset`' because you nearly always want to get the colon-separated version (`$TEXINPUTS` here) visible to the environment, and I've set `-U` as described above for `$path` because it's a neat feature anyway. Now you can assign to the

array `$texinputs` and let the programme (TeX or its derivatives) see `$TEXINPUTS`. Another useful variable to do this with is `$LD_LIBRARY_PATH`, which on most modern versions of UNIX (and Linux) tells the system where to find the libraries which provide extra functions when it runs a programme.

2.5.14 Version-specific things

Since zsh changes faster than almost any other command interpreter known to humankind, you will often find you need to find out what version you are using. This can get a bit verbose; indeed, the parameter you need to check, which is now `$ZSH_VERSION`, used simply to be called `$VERSION` before version `3.0`. If you are not using legacy software of that kind, you can probably get away with tests like this:

```
if [[ $ZSH_VERSION == 3.1.<5->* ||
      $ZSH_VERSION == 3.<2->* ||
      $ZSH_VERSION == <4->* ]]; then
  # set feature which appeared first in 3.1.5
fi
```

It's like that to be futureproof: it says that if this is a 3.1 release, it has to be at least 3.1.5, but any 3.2 release (there weren't any), or any release 4 or later, will also be OK. The '`<5->`' etc. are advanced pattern matching tests: pattern matching uses the same symbols as globbing, but to test other things, here what's on the left of the '`==`'. This one matches any number which is at least 5, for example 6 or 10 or 252, but not 1 or 4. There are also development releases; nowadays the version numbers look like *X*.*Y*.*Z*-*tag*-*N* (*tag* is some short word, the others are numbers) but unless you're keeping up with development you won't need to look for those, since they aren't released officially. That '`==`' in the test could also be just '`=`', but the manual says the former is preferred, so I've used them here, even though people usually don't bother.

Version 4 of zsh provides a function `is-at-least` to do this for you: it looks only at the numbers *X*, *Y* and *Z* (and *N* if it exists), ignoring all letters and punctuation. You give it the minimum version of the shell you need and it returns true if the current shell is recent enough. For example, '`is-at-least 3.1.6-pws-9`' will return true if the current version of zsh is 3.1.6-dev-20 (or 3.1.9, or 4.0.1, and so on), which is the correct behaviour. As with any other shell function, you have to arrange for `is-at-least` to be `autoload`ed if you want to use it.

2.5.15 Everything else

There are many other possibilities for things to go in startup files; in particular, I haven't touched on defining things for the line editor and setting up completion. There's quite a lot to explain for those, so I'll come back to those in the appropriate chapters. You just need to remember that all that stuff should go in `.zshrc`, since you need it for all interactive shells, and for no others.

Chapter 3

Dealing with basic shell syntax

This chapter is a more thorough examination of much of what appeared in the chapter 2; to be more specific, I assume you're sitting in front of your terminal about to use the features you just set up in your initialisation files and want to know enough to get them going. Actually, you will probably spend most of the time editing command lines and in particular completing commands — both of these activities are covered in later chapters. For now I'm going to talk about commands and the syntax that goes along with using them. This will let you write shell functions and scripts to do more of your work for you.

In the following there are often several consecutive paragraphs about quite minor features. If you find you read this all through the first time, maybe you need to get out more. Most people will probably find it better to skim through to find what the subject matter is, then come back if they later find they want to know more about a particular aspect of the shell's commands and syntax.

One aspect of the syntax is left to chapter 5: there's just so much to it, and it can be so useful if you know enough to get it right, that it can't all be squashed in here. The subject is expansion, covering a multitude of things such as parameter expansion, globbing and history expansions. You've already met the basics of these in chapter 2; but if you want to know how to pick a particular file with a globbing expression with pinpoint accuracy, or how to make a single parameter expansion reduce a long expression to the words you need, you should read that chapter; it's more or less self-contained, so you don't necessarily need to know everything in this one.

We start with the most basic issue in any command line interpreter, running commands. As you know, you just type words separated by spaces, where the first word is a command and the remainder are arguments to it. It's important to distinguish between the types of command.

3.1 External commands

External commands are the easiest, because they have the least interaction with the shell — many of the commands provided by the shell itself, which are described in the next section, are built into the shell especially to avoid this difficulty.

The only major issue is therefore how to find them. This is done through the parameters `$path` and `$PATH`, which, as I described in chapter 2, are tied together because although

the first one is more useful inside the shell — being an array, its various parts can be manipulated separately — the second is the one that is used by other commands called by the shell; in the jargon, `$PATH` is 'exported to the environment', which means exactly that other commands called by the shell can see its value.

So suppose your `$path` contains

```
/home/pws/bin /usr/local/bin /bin /usr/bin
```

and you try to run '`ls`'. The shell first looks in `/home/pws/bin` for a command called `ls`, then in `/usr/local/bin`, then in `/bin`, where it finds it, so it executes `/bin/ls`. Actually, the operating system itself knows about paths if you execute a command the right way, so the shell doesn't strictly need to.

There is a subtlety here. The shell tries to remember where the commands are, so it can find them again the next time. It keeps them in a so-called 'hash table', and you find the word 'hash' all over the place in the documentation: all it means is a fast way of finding some value, given a particular key. In this case, given the name of a command, the shell can find the path to it quickly. You can see this table, in the form '*key*=*value*', by typing '`hash`'.

In fact the shell only does this when the option `HASH_CMDS` is set, as it is by default. As you might expect, it stops searching when it finds the directory with the command it's looking for. There is an extra optimisation in the option `HASH_ALL`, also set by default: when the shell scans a directory to find a command, it will add all the other commands in that directory to the hash table. This is sensible because on most UNIX-like operating systems reading a whole lot of files in the same directory is quite fast.

The way commands are stored has other consequences. In particular, zsh won't look for a new command if it already knows where to find one. If I put a new `ls` command in `/usr/local/bin` in the above example, zsh would continue to use `/bin/ls` (assuming it had already been found). To fix this, there is the command `rehash`, which actually empties the command hash table, so that finding commands starts again from scratch. Users of csh may remember having to type `rehash` quite a lot with new commands: it's not so bad in zsh, because if no command was already hashed, or the existing one disappeared, zsh will automatically scan the path again; furthermore, zsh performs a `rehash` of its own accord if `$path` is altered. So adding a new duplicate command somewhere towards the head of `$path` is the main reason for needing `rehash`.

One thing that can happen if zsh hasn't filled its command hash table and so doesn't know about all external commands is that the `AUTO_CD` option, mentioned in the previous chapter and again below, can think you are trying to change to a particular directory with the same name as the command. This is one of the drawbacks of `AUTO_CD`.

To be a little bit more technical, it's actually not so obvious that command hashing is needed at all; many modern operating systems can find commands quickly without it. The clincher in the case of zsh is that the same hash table is necessary for command completion, a very commonly used feature. If you type '`compr<TAB>`', the shell completes this to '`compress`'. It can only do this if it has a list of commands to complete, and this is the hash table. (In this case it didn't need to know where to find the command, just its name, but it's only a little extra work to store that too.) If you were following the previous paragraphs, you'll realise zsh doesn't necessarily know *all* the possible commands at the time you hit `TAB`, because it only looks when it needs to. For this purpose, there is another option, `HASH_LIST_ALL`, again set by default, which will make sure the command hash table is full when you try to complete a command. It only needs to do this once (unless you alter `$path`), but it does mean the first command completion is slow. If `HASH_LIST_ALL` is not set, command completion is not

available: the shell could be rewritten to search the path laboriously every single time you try to complete a command name, but it just doesn't seem worth it.

The fact that `$PATH` is passed on from the shell to commands called from it (strictly only if the variable is marked for export, as it usually is — this is described in more detail with the `typeset` family of builtin commands below) also has consequences. Some commands call subcommands of their own using `$PATH`. If you have that set to something unusual, so that some of the standard commands can't be found, it could happen that a command which *is* found nonetheless doesn't run properly because it's searching for something it can't find in the path passed down to it. That can lead to some strange and confusing error messages.

One important thing to remember about external commands is that the shell continues to exist while they are running; it just hangs around doing nothing, waiting for the job to finish (though you can tell it not to, as we'll see). The command is given a completely new environment in which to run; changes in that don't affect the shell, which simply starts up where it left off after the command has run. So if you need to do something which changes the state of the shell, an external command isn't good enough. This brings us to builtin commands.

3.2 Builtin commands

Builtin commands, or builtins for short, are commands which are part of the shell itself. Since builtins are necessary for controlling the shell's own behaviour, introducing them actually serves as an introduction to quite a lot of what is going on in the shell. So a fair fraction of what would otherwise appear later in the chapter has accumulated here, one way or another. This does make things a little tricksy in places; count how many times I use the word '`subtle`' and keep it for your grandchildren to see.

I just described one reason for builtins, but there's a simpler one: speed. Going through the process of setting up an entirely new environment for the command at the beginning, swapping between this command and anything else which is being run on the computer, then destroying it again at the end is considerable overkill if all you want to do is, say, print out a message on the screen. So there are builtins for this sort of thing.

3.2.1 Builtins for printing

The commands '`echo`' and '`print`' are shell builtins; they just show what you typed, after the shell has removed all the quoting. The difference between the two is really historical: '`echo`' came first, and only handled a few simple options; ksh provided '`print`', which had more complex options and so became a different command. The difference remains between the two commands in zsh; if you want wacky effects, you should look to `print`. Note that there is usually also an external command called `echo`, which may not be identical to zsh's; there is no standard external command called `print`, but if someone has installed one on your system, the chances are it sends something to the printer, not the screen.

One special effect is '`print -z`' puts the arguments onto the editing buffer stack, a list maintained by the shell of things you are about to edit. Try:

```
print -z print -z print This is a line
```

(it may look as if something needs quoting, but it doesn't) and hit return three times. The

first time caused everything after the first '`print -z`' to appear for you to edit, and so on.

For something more useful, you can write functions that give you a line to edit:

```
fn() { print -z print The time now is $(date); }
```

Now when you type '`fn`', the line with the date appears on the command line for you to edit. The option '`-s`' is a bit similar; the line appears in the history list, so you will see it if you use up-arrow, but it doesn't reappear automatically.

A few other useful options, some of which you've already seen, are

-r don't interpret special character sequences like '`\n`'

-P use '`%`' as in prompts

-n don't put a newline at the end in case there's more output to follow

-c print the output in columns — this means that '`print -c *`' has the effect of a sort of poor person's '`ls`', only faster

-l use one line per argument instead of one column, which is sometimes useful for sticking lists into files, and for working out what part of an array parameter is in each element.

If you don't use the `-r` option, there are a whole lot of special character sequences. Many of these may be familiar to you from C.

\n newline

\t tab

\e or \E escape character

\a ring the bell (alarm), usually a euphemism for a hideous beep

\b move back one character.

\c don't print a newline — like the `-n` option, but embedded in the string. This alternative comes from Berkeley UNIX.

\f form feed, the phrase for 'advance to next page' from the days when terminals were called teletypes, maybe more familiar to you as `^L`

\r carriage return — when printed, the annoying `^M`'s you get in DOS files, but actually rather useful with '`print`', since it will erase everything to the start of the line. The combination of the `-n` option and a `\r` at the start of the print string can give the illusion of a continuously changing status line.

\v vertical tab, which I for one have never used (I just tried it now and it behaved like a newline, only without assuming a carriage return, but that's up to your terminal).

In fact, you can get any of the 255 characters possible, although your terminal may not like some or all of the ones above 127, by specifying a number after the backslash. Normally this consists of three octal characters, but you can use two hexadecimal characters after `\x` instead — so '`\n`', '`\012`' and '`\x0a`' are all newlines. '`\`' itself escapes any other character, i.e. they appear as themselves even if they normally wouldn't.

Two notes: first, don't get confused because '`n`' is the fourteenth letter of the alphabet; printing '`\016`' (fourteen in octal) won't do you any good. The remedy, after you discover your text is unreadable (for VT100-like terminals including xterm), is to print '`\017`'.

Secondly, those backslashes can land you in real quoting difficulties. Normally a backslash on the command line escapes the next character — this is a *different* form of escaping to `print`'s — so

```
print \n
```

doesn't produce a newline, it just prints out an '`n`'. So you need to quote that. This means

```
print \\
```

passes a single backslash to quote, and

```
print \\n
```

or

```
print '\n'
```

prints a newline (followed by the extra one that's usually there). To print a real backslash, you would thus need

```
print \\\\
```

Actually, you can get away with the two if there's nothing else after — `print` just shrugs its shoulders and outputs what it's been given — but that's not a good habit to get into. There are other ways of doing this: since single quotes quote anything, including backslashes (they are the only way of making backslashes behave like normal characters), and since the '`-r`' option makes print treat characters normally,

```
print -r '\'
```

has the same effect. But you need to remember the two levels of quoting for backslashes. Quotes aren't special to `print`, so

```
print \'
```

is good enough for printing a quote.

echotc

There's an oddity called '`echotc`', which takes as its argument 'termcap' capabilities. This now lives in its own module, `zsh/termcap`.

Termcap is a now rather old-fashioned way of giving the commands necessary for performing various standard operations on terminals: moving the cursor, clearing to the end of the

line, turning on standout mode, and so on. It has now been replaced almost everywhere by 'terminfo', a completely different way of specifying capabilities, and by 'curses', a more advanced system for manipulating objects on a character terminal. This means that the arguments you need to give to `echotc` can be rather hard to come by; try the `termcap` manual page; if there are two, it's probably the one in section five which gives the codes, i.e. '`man 5 zsh`' or '`man -s 5 zsh`' on Solaris. Otherwise you'll have to search the web. The reason the `zsh` manual doesn't give a list is that the shell only uses a few well-known sequences, and there are very many others which will work with `echotc`, because the sequences are interpreted by a the terminal, not the shell.

This chunk gives you a flavour:

```
zmodload -i zsh/termcap
echotc md
echo -n bold
echotc mr
echo -n reverse
echotc me
echo
```

First we make sure the module is loaded into the shell; on some older operating systems, this only works if it was compiled in when zsh was installed. The option `-i` to `zmodload` stops the shell from complaining if the module was already loaded. This is a sensible way of ensuring you have the right facilities available in a shell function, since loading a module makes it available until it is explicitly unloaded.

You should see '`bold`' in bold characters, and '`reverse`' in bold reverse video. The '`md`' capability turns on bold mode; '`mr`' turns on reverse video; '`me`' turns off both modes. A more typical zsh way of doing this is:

```
print -P '%Bbold%Sreverse%b%s'
```

which should show the same thing, but using prompt escapes — prompts are the most common use of special fonts. The '`%S`' is because zsh calls reverse 'standout' mode, because it does. (On a colour xterm, you may find 'bold' is interpreted as 'blue'.)

There's a lot more you can do with `echotc` if you really try. The shell has just acquired a way of printing terminfo sequences, predictably called `echoti`, although it's only available on systems where zsh needs terminfo to compile — this happens when the termcap code is actually a part of terminfo. The good news about this is that terminfo tends to be better documented, so you have a good chance of finding out the capabilities you want from the `terminfo` manual page. The `echoti` command lives in another predictably named module, `zsh/terminfo`.

3.2.2 Other builtins just for speed

There are only a few other builtins which are there just to make things go faster. Strictly, tests could go into this category, but as I explained in the last chapter it's useful to have tests in the form

```
if [[ $var1 = $var2 ]]; then
  print doing something
```

```
fi
```

be treated as a special syntax by the shell, in case `$var1` or `$var2` expands to nothing which would otherwise confuse it. This example consists of two features described below: the test itself, between the double square brackets, which is true if the two substituted values are the same string, and the '`if`' construct which runs the commands in the middle (here just the `print`) if that test was true.

The builtins '`true`' and '`false`' do nothing at all, except return a command status zero or one, respectively. They're just used as placeholders: to run a loop forever — `while` will also be explained in more detail later — you use

```
while true; do
  print doing something over and over
done
```

since the test always succeeds.

A synonym for '`true`' is '`:`'; it's often used in this form to give arguments which have side effects but which shouldn't be used — something like

```
: ${param:=value}
```

which is a common idiom in all Bourne shell derivatives. In the parameter expansion, `$param` is given the value `value` if it was empty before, and left alone otherwise. Since that was the only reason for the parameter expansion, you use `:` to ignore the argument. Actually, the shell blithely builds the command line — the colon, followed by whatever the value of `$param` is, whether or not the assignment happened — then executes the command; it just so happens that '`:`' takes no notice of the arguments it was given. If you're switching from ksh, you may expect certain synonyms like this to be aliases, rather than builtins themselves, but in zsh they are actually builtins; there are no aliases predefined by the shell. (You can still get rid of them using '`disable`', as described below.)

3.2.3 Builtins which change the shell's state

A more common use for builtins is that they change something inside the shell, or report information about what's going on in the shell. There is one vital thing to remember about external commands. It applies, too, to other cases we'll meet where the shell 'forks', literally splitting itself into two parts, where the forked-off part behaves just like an external command. In both of these cases, the command is in a different *process*, UNIX's basic unit of things that run. (In fact, even Windows knows about processes nowadays, although they interact a little bit differently with one another.)

The vital thing is that no change in a separate process started by the shell affects the shell itself. The most common case of this is the current directory — every process has its own current directory. You can see this by starting a new zsh:

```
% pwd               # show the current directory
~
% zsh               # start a new shell, which
                    # is a separate process
```

```
% cd tmp
% pwd               # now I'm in a different
                    # directory...
~/tmp
% exit              # leave the new shell...
% pwd               # now I'm back where I was...
~
```

Hence the `cd` command must be a shell builtin, or this would happen every time you ran it.

Here's a more useful example. Putting parentheses around a command asks the shell to start a different process for it. That's useful when you specifically *don't* want the effects propagating back:

```
(cd some-other-dir; run-some-command)
```

runs the command, but doesn't change the directory the 'real' shell is in, only its forked-off 'subshell'. Hence,

```
% pwd
~
% (cd /; pwd)
/
% pwd
~
```

There's a more subtle case:

```
cd some-other-dir | print Hello
```

Remember, the '`|`' ('pipe') connects the output of the first command to the input of the next — though actually no information is passed that way in this example. In zsh, all but the last portion of the 'pipeline' thus created is run in different processes. Hence the `cd` doesn't affect the main shell. I'll refer to it as the 'parent' shell, which is the standard UNIX language for processes; when you start another command or fork off a subshell, you are creating 'children' (without meaning to be morbid, the children usually die first in this case). Thus, as you would guess,

```
print Hello | cd some-other-dir
```

does have the effect of changing the directory. Note that other shells do this differently; it is always guaranteed to work this way in zsh, because many people rely on it for setting parameters, but many shells have the *left* hand of the pipeline being the bit that runs in the parent shell. If both sides of the pipe symbol are external commands of some sort, both will of course run in subprocesses.

There are other ways you change the state of the shell, for example by declaring parameters of a particular type, or by telling it how to interpret certain commands, or, of course, by changing options. Here are the most useful, grouped in a vaguely logical fashion.

3.2.4 cd and friends

You will not by now be surprised to learn that the 'cd' command changes directory. There is a synonym, 'chdir', which as far as I know no-one ever uses. (It's the same name as the system call, so if you had been programming in C or Perl and forgot that you were now using the shell, you might use 'chdir'. But that seems a bit far-fetched.)

There are various extra features built into cd and chdir. First, if you miss out the directory to which you want to change, you will be taken to your home directory, although it's not as if 'cd ~' is all that hard to type.

Next, the command 'cd -' is special: it takes you to the last directory you were in. If you do a sequence of cd commands, only the immediately preceding directory is remembered; they are not stacked up.

Thirdly, there is a shortcut for changing between similarly named directories. If you type 'cd <old> <new>', then the shell will look for the first occurrence of the string '<old>' in the current directory, and try to replace it with '<new>'. For example,

```
% pwd
~/src/zsh-3.0.8/Src
% cd 0.8 1.9
~/src/zsh-3.1.9/Src
```

The cd command actually reported the new directory, as it usually does if it's not entirely obvious where it's taken you.

Note that only the *first* match of <old> is taken. It's an easy mistake to think you can change from /home/export1/pws/mydir1/something to /home/export1/pws/mydir2/something with 'cd 1 2', but that first '1' messes it up. Arguably the shell could be smarter here. Of course, 'cd r1 r2' will work in this case.

cd's friend 'pwd' (print working directory) tells you what the current working directory is; this information is also available in the shell parameter $PWD, which is special and automatically updated when the directory changes. Later, when you know all about expansion, you will find that you can do tricks with this to refer to other directories. For example, ${PWD/old/new} uses the parameter substitution mechanism to refer to a different directory with old replaced by new — and this time old can be a pattern, i.e. something with wildcard matches in it. So if you are in the zsh-3.0.8/Src directory as above and want to copy a file from the zsh-3.1.9/Src directory, you have a shorthand:

```
cp ${PWD/0.8/1.9}/myfile.c .
```

Symbolic links

Zsh tries to track directories across symbolic links. If you're not familiar with these, you can think of them as a filename which behaves like a pointer to another file (a little like Windows' shortcuts, though UNIX has had them for much longer and they work better). You create them like this (ln is not a builtin command, but its use to make symbolic links is very standard these days):

```
ln -s existing-file-name name-of-link
```

for example

```
ln -s /usr/bin/ln ln
```

creates a file called `ln` in the current directory which does nothing but point to the file `/usr/bin/ln`. Symbolic links are very good at behaving as much like the original file as you usually want; for example, you can run the `ln` link you've just created as if it were `/usr/bin/ln`. They show up differently in a long file listing with '`ls -l`', the last column showing the file they point to.

You can make them point to any sort of file at all, including directories, and that is why they are mentioned here. Suppose you create a symbolic link from your home directory to the root directory and change into it:

```
ln -s / ~/mylink
cd ~/mylink
```

If you don't know it's a link, you expect to be able to change to the parent directory by doing '`cd ..`'. However, the operating system — which just has one set of directories starting from `/` and going down, and ignores symbolic links after it has followed them, they really are just pointers — thinks you are in the root directory `/`. This can be confusing. Hence zsh tries to keep track of where *you* probably think you are, rather than where the system does. If you type '`pwd`', you will see '`/home/you/mylink`' (wherever your home directory is), not '`/`'; if you type '`cd ..`', you will find yourself back in your home directory.

You can turn all this second-guessing off by setting the option `CHASE_LINKS`; then '`cd ~/mydir; pwd`' will show you to be in `/`, where changing to the parent directory has no effect; the parent of the root directory is the root directory, except on certain slightly psychedelic networked file systems. This does have advantages: for example, '`cd ~/mydir; ls ..`' always lists the root directory, not your home directory, regardless of the option setting, because `ls` doesn't know about the links you followed, only zsh does, and it treats the `..` as referring to the root directory. Having `CHASE_LINKS` set allows '`pwd`' to warn you about where the system thinks you are.

An aside for non-UNIX-experts (over 99.9% of the population of the world at the last count): I said 'symbolic links' instead of just 'links' because there are others called 'hard links'. This is what '`ln`' creates if you don't use the `-s` option. A hard link is not so much a pointer to a file as an alternative name for a file. If you do

```
ln myfile othername
ls -l
```

where `myfile` already exists you can't tell which of `myfile` and `othername` is the original — and in fact the system doesn't care. You can remove either, and the other will be perfectly happy as the name for the file. This is pretty much how renaming files works, except that creating the hard link is done for you in that case. Hard links have limitations — you can't link to directories, or to a file on another disk partition (and if you don't know what a disk partition is, you'll see what a limitation that can be). Furthermore, you usually want to know which is the original and which is the link — so for most users, creating symbolic links is more useful. The only drawback is that following the pointers is a tiny bit slower; if you think you can notice the difference, you definitely ought to slow down a bit.

The target of a symbolic link, unlike a hard link, doesn't actually have to exist and no checking is performed until you try to use the link. The best thing to do is to run '`ls -lL`'

when you create the link; the `-L` part tells `ls` to follow links, and if it worked you should see that your link is shown as having exactly the same characteristics as the file it points to. If it is still shown as a link, there was no such file.

While I'm at it, I should point out one slight oddity with symbolic links: the name of the file linked to (the first name), if it is not an absolute path (beginning with `/` after any `~` expansion), is treated relative to the directory where the link is created — not the current directory when you run `ln`. Here:

```
ln -s ../mydir ~/links/otherdir
```

the link `otherdir` will refer to `mydir` in *its own* parent directory, i.e. `~/links` — not, as you might think, the parent of the directory where you were when you ran the command. What makes it worse is that the second word, if is not an absolute path, *is* interpreted relative to the directory where you ran the command.

$cdpath and AUTO_CD

We're nowhere near the end of the magic you can do with directories yet (and, in fact, I haven't even got to the zsh-specific parts). The next trick is `$cdpath` and `$CDPATH`. They look a lot like `$path` and `$PATH` which you met in the last chapter, and I mentioned them briefly back in the last chapter in that context: `$cdpath` is an array of directories, while `$CDPATH` is colon-separated list behaving otherwise like a scalar variable. They give a list of directories whose subdirectories you may want to change into. If you use a normal cd command (i.e. in the form '`cd` *dirname*', and *dirname* does not begin with a `/` or `~`, the shell will look through the directories in `$cdpath` to find one which contains the subdirectory *dirname*. If `$cdpath` isn't set, as you'd guess, it just uses the current directory.

Note that `$cdpath` is always searched in order, and you can put a `.` in it to represent the current directory. If you do, the current directory will always be searched *at that point*, not necessarily first, which may not be what you expect. For example, let's set up some directories:

```
mkdir ~/crick ~/crick/dna
mkdir ~/watson ~/watson/dna
cdpath=(~/crick .)
cd ~/watson
cd dna
```

So I've moved to the directory `~/watson`, which contains the subdirectory `dna`, and done '`cd dna`'. But because of `$cdpath`, the shell will look first in `~/crick`, and find the `dna` there, and take you to that copy of the self-reproducing directory, not the one in `~/watson`. Most people have `.` at the start of their `cdpath` for that reason. However, at least `cd` warns you — if you tried it, you will see that it prints the name of the directory it's picked in cases like this.

In fact, if you don't have `.` in your directory at all, the shell will always look there first; there's no way of making `cd` never change to a subdirectory of the current one, short of turning `cd` into a function. Some shells don't do this; they use the directories in `$cdpath`, and only those.

There's yet another shorthand, this time specific to zsh: the option `AUTO_CD` which I mentioned in the last chapter. That way a command without any arguments which is really

a directory will take you to that directory. Normally that's perfect — you would just get a 'command not found' message otherwise, and you might as well make use of the option. Just occasionally, however, the name of a directory clashes with the name of a command, builtin or external, or a shell function, and then there can be some confusion: zsh will always pick the command as long as it knows about it, but there are cases where it doesn't, as I described above.

What I didn't say in the last chapter is that `AUTO_CD` respects `$cdpath`; in fact, it really is implemented so that '*dirname*' on its own behaves as much like '`cd` *dirname*' as is possible without tying the shell's insides into knots.

The directory stack

One very useful facility that zsh inherited from the C-shell family (traditional Korn shell doesn't have it) is the directory stack. This is a list of directories you have recently been in. If you use the command '`pushd`' instead of '`cd`', e.g. '`pushd` *dirname*', then the directory you are in is saved in this list, and you are taken to *dirname*, using `$CDPATH` just as `cd` does. Then when you type '`popd`', you are taken back to where you were. The list can be as long as you like; you can `pushd` any number of directories, and each `popd` will take you back through the list (this is how a 'stack', or more precisely a 'last-in-first-out' stack usually operates in computer jargon, hence the name 'directory stack').

You can see the list — which always starts with the current directory — with the `dirs` command. So, for example:

```
cd ~
pushd ~/src
pushd ~/zsh
dirs
```

displays

```
~/zsh ~/src ~
```

and the next `popd` will take you back to `~/src`. If you do it, you will see that `pushd` reports the list given by `dirs` automatically as it goes along; you can turn this off with the option `PUSHD_SILENT`, when you will have to rely on typing `dirs` explicitly.

In fact, a lot of the use of this comes not from using simple `pushd` and `popd` combinations, but from two other features. First, '`pushd`' on its own swaps the top two directories on the stack. Second, `pushd` with a numeric argument preceded by a '`+`' or '`-`' can take you to one of the other directories in the list. The command '`dirs -v`' tells you the numbers you need; `0` is the current directory. So if you get,

```
0       ~/zsh
1       ~/src
2       ~
```

then '`pushd +2`' takes you to `~`. (A little suspension of disbelief that I didn't just use `AUTO_CD` and type '`..`' is required here.) If you use a `-`, it counts from the other end of the list; `-0` (with apologies to the numerate) is the last item, i.e. the same as `~` in this case. Some

people are used to having the '–' and '+' arguments behave the other way around; the option PUSHD_MINUS exists for this.

Apart from PUSHD_SILENT and PUSHD_MINUS, there are a few other relevant options. Setting PUSHD_IGNORE_DUPS means that if you pushd to a directory which is already somewhere in the list, the duplicate entry will be silently removed. This is useful for most human operations — however, if you are using pushd in a function or script to remember previous directories for a future matching popd, this can be dangerous and you probably want to turn it off locally inside the function.

AUTO_PUSHD means that any directory-changing command, including an auto-cd, is treated as a pushd command with the target directory as argument. Using this can make the directory stack get very long, and there is a parameter $DIRSTACKSIZE which you can set to specify a maximum length. The oldest entry (the highest number in the 'dirs -v' listing) is automatically removed when this length is exceeded. There is no limit unless this is explicitly set.

The final pushd option is PUSHD_TO_HOME. This makes pushd on its own behave like cd on its own in that it takes you to your home directory, instead of swapping the top two directories. Normally a series of 'pushd' commands works pretty much like a series of 'cd -' commands, always taking you the directory you were in before, with the obvious difference that 'cd -' doesn't consult the directory stack, it just remembers the previous directory automatically, and hence it can confuse pushd if you just use 'cd -' instead.

There's one remaining subtlety with pushd, and that is what happens to the rest of the list when you bring a particular directory to the front with something like 'pushd +2'. Normally the list is simply cycled, so the directories which were +3, and +4 are now right behind the new head of the list, while the two directories which were ahead of it get moved to the end. If the list before was:

```
dir1  dir2  dir3  dir4
```

then after pushd +2 you get

```
dir3  dir4  dir1 dir2
```

That behaviour changed during the lifetime of zsh, and some of us preferred the old behaviour, where that one directory was yanked to the front and the rest just closed the gap:

```
# Old behaviour
dir3  dir1  dir2  dir4
```

so that after a while you get a 'greatest hits' group at the front of the list. If you like this behaviour too (I feel as if I'd need to have written papers on group theory to like the new behaviour) there is a function pushd supplied with the source code, although it's short enough to repeat here — this is in the form for autoloading in the zsh fashion:

```
# pushd function to emulate the old zsh behaviour.
# With this, pushd +/-n lifts the selected element
# to the top of the stack instead of cycling
# the stack.
```

```
emulate -R zsh
setopt localoptions

if [[ ARGC -eq 1 && "$1" == [+-]<-> ]] then
        setopt pushdignoredups
        builtin pushd ~$1
else
        builtin pushd "$@"
fi
```

The '`&&`' is a logical 'and', requiring both tests to be true. The tests are that there is exactly one argument to the function, and that it has the form of a '+' or a '–' followed by any number ('`<->`' is a special zsh pattern to match any number, an extension of forms like '`<1-100>`' which matches any number in the range 1 to 100 inclusive).

Referring to other directories

Zsh has two ways of allowing you to refer to particular directories. They have in common that they begin with a ~ (in very old versions of zsh, the second form actually used an '=', but the current way is much more logical).

You will certainly be aware, because I've made a lot of use of it, that a '~' on its own or followed by a / refers to your own home directory. An extension of this — again from the C-shell, although the Korn shell has it too in this case — is that `~name` can refer to the home directory of any user on the system. So if your user name is `pws`, then ~ and `~pws` are the same directory.

Zsh has an extension to this; you can actually name your own directories. This was described in chapter 2, à propos of prompts, since that is the major use:

```
host% PS1='%~? '
~? cd zsh/Src
~/zsh/Src? zsrc=$PWD
~/zsh/Src? echo ~zsrc
/home/pws/zsh/Src
~zsrc?
```

Consult that chapter for the ways of forcing a parameter to be recognised as a named directory.

There's a slightly more sophisticated way of doing this directly:

```
hash -d zsrc=~/zsh/Src
```

makes `~zsrc` appear in prompts as before, and in this case there is no parameter `$zsrc`. This is the purist's way (although very few zsh users are purists). You can guess what '`unhash -d zsrc`' does; this works with directories named via parameters, too, but leaves the parameter itself alone.

It's possible to have a named directory with the same name as a user. In that case '`~name`' refers to the directory you named explicitly, and there is no easy way of getting `name`'s home directory without removing the name you defined.

If you're using named directories with one of the `cd`-like commands or `AUTO_CD`, you can set the option `CDABLEVARS` which allows you to omit the leading `~`; '`cd zsrc`' with this option would take you to `~zsrc`. The name is a historical artifact and now a misnomer; it really is named directories, not parameters (i.e. variables), which are used.

The second way of referring to directories with `~`'s is to use numbers instead of names: the numbers refer to directories in the directory stack. So if `dirs -v` gives you

```
0       ~zsf
1       ~src
```

then `~+1` and `~-0` (not very mathematical, but quite logical if you think about it) refer to `~src`. In this case, unlike pushd arguments, you can omit the `+` and use `~1`. The option `PUSHD_MINUS` is respected. You'll see this was used in the `pushd` function above: the trick was that `~+3`, for example, refers to the same element as `pushd +3`, hence `pushd ~+3` pushed that directory onto the front of the list. However, we set `PUSHD_IGNORE_DUPS`, so that the value in the old position was removed as well, giving us the effect we wanted of simply yanking the directory to the front with no trick cycling.

3.2.5 Command control and information commands

Various builtins exist which control how you access commands, and which show you information about the commands which can be run.

The first two are strictly speaking 'precommand modifiers' rather than commands: that means that they go before a command line and modify its behaviour, rather than being commands in their own right. If you put '`command`' in front of a command line, the command word (the next one along) will be taken as the name of an external command, however it would normally be interpreted; likewise, if you put '`builtin`' in front, the shell will try to run the command as a builtin command. Normally, shell functions take precedence over builtins which take precedence over external commands. So, for example, if your printer control system has the command '`enable`' (as many System V versions do), which clashes with a builtin I am about to talk about, you can run '`command enable lp`' to enable a printer; otherwise, the builtin enable would have been run. Likewise, if you have defined `cd` to be a function, but this time want to call the normal builtin `cd`, you can say '`builtin cd mydir`'.

A common use for `command` is inside a shell function of the same name. Sometimes you want to enhance an ordinary command by sticking some extra stuff around it, then calling that command, so you write a shell function of the same name. To call the command itself inside the shell function, you use '`command`'. The following works, although it's obviously not all that useful as it stands:

```
ls() {
  command ls "$[@]"
}
```

so when you run '`ls`', it calls the function, which calls the real `ls` command, passing on the arguments you gave it.

You can gain longer lasting control over the commands which the shell will run with the '`disable`' and '`enable`' commands. The first normally takes builtin arguments; each such

builtin will not be recognised by the shell until you give an 'enable' command for it. So if you want to be able to run the external enable command and don't particularly care about the builtin version, 'disable enable' (sorry if that's confusing) will do the trick. Ha, you're thinking, you can't run 'enable enable'. That's correct: some time in the dim and distant past, builtin enable enable' would have worked, but currently it doesn't; this may change, if I remember to change it. You can list all disabled builtins with just 'disable' on its own — most of the builtins that do this sort of manipulation work like that.

You can manipulate other sets of commands with disable and enable by giving different options: aliases with the option -a, functions with -f, and reserved words with -r. The first two you probably know about, and I'll come to them anyway, but 'reserved words' need describing. They are essentially builtin commands which have some special syntactic meaning to the shell, including some symbols such as '{' and '[['. They take precedence over everything else except aliases — in fact, since they're syntactically special, the shell needs to know very early on that it has found a reserved word, it's no use just waiting until it tries to execute a command. For example, if the shell finds '[[' it needs to know that everything until ']]' must be treated as a test rather than as ordinary command arguments. Consequently, you wouldn't often want to disable a reserved word, since the shell wouldn't work properly. The most obvious reason why you might would be for compatibility with some other shell which didn't have one. You can get a complete list with:

```
whence -wm '*' | grep reserved
```

which I'll explain below, since I'm coming to 'whence'.

Furthermore, I tend to find that if I want to get rid of aliases or functions I use the commands 'unalias' and 'unfunction' to get rid of them permanently, since I always have the original definitions stored somewhere, so these two options may not be that useful either. Disabling builtins is definitely the most useful of the four possibilities for disable.

External commands have to be manipulated differently. The types given above are handled internally by the shell, so all it needs to do is remember what code to call. With external commands, the issue instead is how to find them. I mentioned rehash above, but didn't tell you that the hash command, which you've already seen with the -d option, can be used to tell the shell how to find an external command:

```
hash foo=/path/to/foo
```

makes foo execute the command using the path shown (which doesn't even have to end in 'foo'). This is rather like an alias — most people would probably do this with an alias, in fact — although a little faster, though you're unlikely to notice the difference. You can remove this with unhash. One gotcha here is that if the path is rehashed, either by calling rehash or when you alter $path, the entire hash table is emptied, including anything you put in in this way; so it's not particularly useful.

In the midst of all this, it's useful to be able to find out what the shell thinks a particular command name does. The command 'whence' tells you this; it also exists, with slightly different options, under the names where, which and type, largely to provide compatibility with other shells. I'll just stick to whence.

Its standard output isn't actually sparklingly interesting. If it's a command somehow known to the shell internally, it gets echoed back, with the alias expanded if it was an alias; if it's an external command it's printed with the full path, showing where it came from; and if it's not known the command returns status 1 and prints nothing.

You can make it more useful with the `-v` or `-c` options, which are more verbose; the first prints out an information message, while the second prints out the definitions of any functions it was asked about (this is also the effect of using '`which`' instead of '`whence`). A very useful option is `-m`, which takes any arguments as patterns using the usual zsh pattern format, in other words the same one used for matching files. Thus

```
whence -vm "*"
```

prints out every command the shell knows about, together with what it thinks of it.

Note the quotes around the '`*`' — you have to remember these anywhere where the pattern is not to be used to generate filenames on the command line, but instead needs to be passed to the command to be interpreted. If this seems a rather subtle distinction, think about what would happen if you ran

```
# Oops.  Better not try this at home.
# (Even better, don't do it at work either.)
whence -vm *
```

in a directory with the files '`foo`' and (guess what) '`bar`' in it. The shell hasn't decided what command it's going to run when it first looks at the command line; it just sees the '`*`' and expands the line to

```
whence -vm foo bar
```

which isn't what you meant.

There are a couple of other tricks worth mentioning: `-p` makes the shell search your path for them, even if the name is matched as something else (say, a shell function). So if you have `ls` defined as a function,

```
which -p ls
```

will still tell what '`command ls`' would find. Also, the option `-a` searches for all commands; in the same example, this would show you both the `ls` command and the `ls` function, whereas `whence` would normally only show the function because that's the one that would be run. The `-a` option also shows if it finds more than one external command in your path.

Finally, the option `-w` is useful because it identifies the type of a command with a single word: `alias`, `builtin`, `command`, `function`, `hashed`, `reserved` or `none`. Most of those are obvious, with `command` being an ordinary external command; `hashed` is an external command which has been explicitly given a path with the `hash` builtin, and `none` means it wasn't recognised as a command at all. Now you know how we extracted the reserved words above.

A close relative of `whence` is `functions`, which applies, of course, to shell functions; it usually lists the definitions of all functions given as arguments, but its relatives (of which `autoload` is one) perform various other tricks, to be described in the section on shell functions below. Be careful with `function`, without the 's', which is completely different and not like `command` or `builtin` — it is actually a keyword used to *define* a function.

3.2.6 Parameter control

There are various builtins for controlling the shells parameters. You already know how to set and use parameters, but it's a good deal more complicated than that when you look at the details.

Local parameters

The principal command for manipulating the behaviour of parameters is '`typeset`'. Its easiest usage is to declare a parameter; you just give it a list of parameter names, which are created as scalar parameters. You can create parameters just by assigning to them, but the major point of '`typeset`' is that if a parameter is created that way inside a function, the parameter is restored to its original value, or removed if it didn't previously exist, at the end of the function — in other words, it has 'local scope' like the variables which you declare in most ordinary programming languages. In fact, to use the jargon it has 'dynamical' rather than 'syntactic' scope, which means that the same parameter is visible in any function called within the current one; this is different from, say, C or FORTRAN where any function or subroutine called wouldn't see any variable declared in the parent function.

The following makes this more concrete.

```
var='Original value'
subfn() {
  print $var
}
fn() {
  print $var
  typeset var='Value in function'
  print $var
  subfn
}
fn
print $var
```

This chunk of code prints out

```
Original value
Value in function
Value in function
Original value
```

The first three chunks of the code just define the parameter `$var`, and two functions, `subfn` and `fn`. Then we call `fn`. The first thing this does is print out `$var`, which gives '`Original value`' since we haven't changed the original definition. However, the `typeset` next does that; as you see, we can assign to the parameter during the typeset. Thus when we print `$var` out again, we get '`Value in function`'. Then `subfn` is called, which prints out the same value as in `fn`, because we haven't changed it — this is where C or FORTRAN would differ, and wouldn't recognise the variable because it hadn't been declared in that function. Finally, `fn` exits and the original value is restored, and is printed out by the final '`print`'.

Note the value changes twice: first at the `typeset`, then again at the end of `fn`. The value of `$var` at any point will be one of those two values.

Although you can do assignments in a `typeset` statement, you can't assign to arrays (I already said this in the last chapter):

```
typeset var=(Doesn\'t work\!)
```

because the syntax with the parentheses is special; it only works when the line consists of nothing but assignments. However, the shell doesn't complain if you try to assign an array to a scalar, or vice versa; it just silently converts the type:

```
typeset var='scalar value'
var=(array value)
```

I put in the assignment in the typeset statement to rub the point in that it creates scalars, but actually the usual way of setting up an array in a function is

```
typeset var
var=()
```

which creates an empty scalar, then converts that to an empty array. Recent versions of the shell have '`typeset -a var`' to do that in one go — but you *still* can't assign to it in the same statement.

There are other catches associated with the fact that `typeset` and its relatives are just ordinary commands with ordinary sets of arguments. Consider this:

```
% typeset var=`echo two words`
% print $var
two
```

What has happened to the '`words`'? The answer is that backquote substitution, to be discussed below, splits words when not quoted. So the `typeset` statement is equivalent to

```
% typeset var=two words
```

There are two ways to get round this; first, use an ordinary assignment:

```
% typeset var
% var=`echo two words`
```

which can tell a scalar assignment, and hence knows not to split words, or quote the backquotes,

```
% typeset var="`echo two words`"
```

There are three important types we haven't talked about; both of these can only be created with `typeset` or one of the similar builtins I'll list in a moment. They are integer types, floating point types, and associative array types.

Numeric parameters

Integers are created with '`typeset -i`', or '`integer`' which is another way of saying the same thing. They are used for arithmetic, which the shell can do as follows:

```
integer i
(( i = 3 * 2 + 1 ))
```

The double parentheses surround a complete arithmetic expression: it behaves as if it's quoted. The expression inside can be pretty much anything you might be used to from arithmetic in other programming languages. One important point to note is that parameters don't need to have the `$` in front, even when their value is being taken:

```
integer i j=12
(( i = 3 * ( j + 4 ) ** 2 ))
```

Here, `j` will be replaced by 12 and `$i` gets the value 768 (sixteen squared times three). One thing you might not recognise is the `**`, which is the 'to the power of' operator which occurs in FORTRAN and Perl. Note that it's fine to have parentheses inside the double parentheses — indeed, you can even do

```
(( i = (3 * ( j + 4 )) ** 2 ))
```

and the shell won't get confused because it knows that any parentheses inside must be in balanced pairs (until you deliberately confuse it with your buggy code).

You would normally use '`print $i`' to see what value had been given to `$i`, of course, and as you would expect it gets printed out as a decimal number. However, `typeset` allows you to specify another base for printing out. If you do

```
typeset -i 16 i
print $i
```

after the last calculation, you should see `16#900`, which means 900 in base 16 (hexadecimal). That's the only effect the option '`-i 16`' has on `$i` — you can assign to it and use it in arithmetical expressions just as normal, but when you print it out it appears in this form. You can use this base notation for inputting numbers, too:

```
(( i = 16#ff * 2#10 ))
```

which means 255 (`ff` in hexadecimal) times 2 (`10` in binary). The shell understands C notation too, so '`16#ff`' could have been expressed '`0xff`'.

Floating point variables are very similar. You can declare them with '`typeset -F`' or '`typeset -E`'. The only difference between the two is, again, on output; `-F` uses a fixed point notation, while `-E` uses scientific (mnemonic: exponential) notation. The builtin '`float`' is equivalent to '`typeset -E`' (because Korn shell does it, that's why). Floating point expressions also work the way you are probably used to:

```
typeset -E e
typeset -F f
(( e = 32/3, f = 32.0/3.0 ))
print $e $f
```

prints

```
1.000000000e+01 10.6666666667
```

Various points: the ',' can separate different expressions, just like in C, so the `e` and `f` assignments are performed separately. The `e` assignment was actually an integer division, because neither 32 nor 3 is a floating point number, which must contain a dot. That means an integer division was done, producing 10, which was then converted to a floating point number only at the end. Again, this is just how grown-up languages work, so it's no use cursing. The `f` assignment was a full floating point performance. Floating point parameters weren't available before version `3.1.7`.

Although this is really a matter for a later chapter, there is a library of floating point functions you can load (actually it's just a way of linking in the system mathematical library). The usual incantation is '`zmodload zsh/mathfunc`'; you may not have 'dynamic loading' of libraries on your system, which may mean that doesn't work. If it does, you can do things like

```
(( pi = 4.0 * atan(1.0) ))
```

Broadly, all the functions which appear in most system mathematical libraries (see the manual page for `math`) are available in zsh.

Like all other parameters created with `typeset` or one of its cousins, integer and floating point parameters are local to functions. You may wonder how to create a global parameter (i.e. one which is valid outside as well as inside the function) which has an integer or floating point value. There's a recent addition to the shell (in version 3.1.6) which allows this: use the flag `-g` to typeset along with any others. For example,

```
fn() {
  typeset -Fg f
  (( f = 42.75 ))
}
fn
print $f
```

If you try it, you will see the value of `$f` has survived beyond the function. The `g` stands for global, obviously, although it's not quite that simple:

```
fn() {
  typeset -Fg f
}
outerfn() {
  typeset f='scalar value'
  fn
  print $f
}
outerfn
```

The function `outerfn` creates a local scalar value for `f`; that's what `fn` sees. So it was not really operating on a 'global' value, it just didn't create a new one for the scope of `fn`. The error message comes because it tried to preserve the value of `$f` while changing its type, and the value wasn't a proper floating point expression. The error message,

```
fn: bad math expression: operator expected at 'value'
```

comes about because assigning to numeric parameters always does an arithmetic evaluation. Operating on '`scalar value`' it found '`scalar`' and assumed this was a parameter, then looked for an operator like '+' to come next; instead it found '`value`'. If you want to experiment, change the string to '`scalar + value`' and set '`value=42`', or whatever, then try again. This is a little confusing (which is a roundabout way of saying it confused me), but consistent with how zsh usually treats parameters.

Actually, to a certain extent you don't need to use the integer and floating point parameters. Any time zsh needs a numeric expression it will force a scalar to the right value, and any time it produces a numeric expression and assigns it to a scalar, it will convert the result to a string. So

```
typeset num=3             # This is the *string* '3'.
(( num = num + 1 ))       # But this works anyway
                          # ($num is still a string).
```

This can be useful if you have a parameter which is sometimes a number, sometimes a string, since zsh does all the conversion work for you. However, it can also be confusing if you always want a number, because zsh can't guess that for you; plus it's a little more efficient not to have to convert back and forth; plus you lose accuracy when you do, because if the number is stored as a string rather than in the internal numeric representation, what you say is what you get (although zsh tends to give you quite a lot of decimal places when converting implicitly to strings). Anyway, I'd recommend that if you know a parameter has to be an integer or floating point value you should declare it as such.

There is a builtin called `let` to handle mathematical expressions, but since

```
let "num = num + 1"
```

is equivalent to

```
(( num = num + 1 ))
```

and the second form is easier and more memorable, you probably won't need to use it. If you do, remember that (unlike BASIC) each mathematical expression should appear as one argument in quotes.

Associative arrays

The one remaining major type of parameter is the associative array; if you use Perl, you may call it a 'hash', but we tend not to since that's really a description of how it's implemented rather than what it does. (All right, what it does is hash things. Now shut up.)

These have to be declared by a typeset statement — there's no getting round it. There are some quite eclectic builtins that produce a filled-in associative array for you, but the only way to tell zsh you want your very own associative array is

```
typeset -A assoc
```

to create `$assoc`. As to what it does, that's best shown by example:

```
typeset -A assoc
assoc=(one eins two zwei three drei)
print ${assoc[two]}
```

which prints 'zwei'. So it works a bit like an ordinary array, but the numeric *subscript* of an ordinary array which would have appeared inside the square bracket is replaced by the string *key*, in this case `two`. The array assignment was a bit deceptive; the 'values' were actually pairs, with 'one' being the key for the value 'eins', and so on. The shell will complain if there are an odd number of elements in such a list. This may also be familiar from Perl. You can assign values one at a time:

```
assoc[four]=vier
```

and also unset one key/value pair:

```
unset 'assoc[one]'
```

where the quotes stop the square brackets from being interpreted as a pattern on the command line.

Expansion has been held over, but you might like to know about the ways of getting back what you put in. If you do

```
print $assoc
```

you just see the values — that's exactly the same as with an ordinary array, where the subscripts 1, 2, 3, etc. aren't shown. Note they are in random order — that's the other main difference from ordinary arrays; associative arrays have no notion of an order unless you explicitly sort them.

But here the keys may be just as interesting. So there is:

```
print ${(k)assoc}
print ${(kv)assoc}
```

giving (if you've followed through all the commands above):

```
four two three
four vier two zwei three drei
```

which print out the keys instead of the values, and the key and value pairs much as you entered them. You can see that, although the order of the pairs isn't obvious, it's the same each time. From this example you can work out how to copy an associative array into another one:

```
typeset -A newass
newass=(${(kv)assoc})
```

where the '(kv)' is important — as is the typeset just before the assignment, otherwise $newass would be a badass ordinary array. You can also prove that ${(v)assoc} does what you would probably expect. There are lots of other tricks, but they are mostly associated with clever types of parameter expansion, to be described in chapter 5.

Other typeset and type tricks

There are variants of typeset, some mentioned sporadically above. There is nothing you can do with any of them that you can't do with typeset — that wasn't always the case; we've tried to improve the orthogonality of the options. They differ in the options which are set by default, and the additional options which are allowed. Here's a list: declare, export, float, integer, local, readonly. I won't confuse you by describing all in detail; see the manual.

If there is an odd one out, it's export, which not only marks a parameter for export but has the -g flag turned on by default, so that that parameter is not local to the function; in other words, it's equivalent to typeset -gx. However, one holdover from the days when the options weren't quite so logical is that typeset -x behaves like export, in other words the -g flag is turned on by default. You can fix this by unsetting the option GLOBAL_EXPORT — the option only exists for compatibility; logically it should always be unset. This is partly because in the old days you couldn't export local parameters, so typeset -x either had to turn on -g or turn off -x; that was fixed for the 3.1.9 release, and (for example) 'local -x' creates a local parameter which is exported to the environment; both the parameter itself, and the value in the environment, will be restored when the function exits. The builtin local is essentially a form of typeset which renounces the -g flag and all its works.

Another old restriction which has gone is that you couldn't make special parameters, in particular $PATH, local to a function; you just modified the original parameter. Now if you say 'typeset PATH', things happen the way you probably expect, with $PATH having its usual effect, and being restored to its old value when the function exits. Since $PATH is still special, though, you should make sure you assign something to it in the function before calling external commands, else it will be empty and no commands will be found. It's possible that you specifically don't want some parameter you make local to have the special property; 3.1.7 and after allow the typeset flag -h to hide the specialness for that parameter, so in 'typeset -h PATH', PATH would be an ordinary variable for the duration of the enclosing function. Internally, the same value as was previously set would continue to be used for finding commands, but it wouldn't be exported.

The second main use of typeset is to set attributes for the parameters. In this case it can operate on an existing parameter, as well as creating a new one. For example,

```
typeset -r msg='This is an important message.'
```

sets the readonly flag (-r) for the parameter msg. If the parameter didn't exist, it would be created with the usual scoping rules; but if it did exist at the current level of scoping, it would be made readonly with the value assigned to it, meaning you can't set that particular copy of the parameter. For obvious reasons, it's normal to assign a value to a readonly parameter when you first declare it. Here's a reality check on how this affects scoping:

```
msg='This is an ordinary parameter'
fn() {
  typeset msg='This is a local ordinary parameter'
  print $msg
```

```
    typeset -r msg='This is a local readonly parameter'
    print $msg
    msg='Watch me cause an error.'
}
fn
print $msg
msg='This version of the parameter'\
' can still be overwritten'
print $msg
```

outputs

```
This is a local ordinary parameter
This is a local readonly parameter
fn:5: read-only variable: msg
This is an ordinary parameter
This version of the parameter can still be overwritten
```

Unfortunately there was a bug with this code until recently — thirty seconds ago, actually: the second `typeset` in `fn` incorrectly added the readonly flag to the existing `msg` *before* attempting to set the new value, which was wrong and inconsistent with what happens if you create a new local parameter. Maybe it's reassuring that the shell can get confused about local parameters, too. (I don't find it reassuring in the slightest, since `typeset` is one of the parts of the code where I tend to fix the bugs, but maybe you do.)

Anyway, when the bug is fixed, you should get the output shown, because the first typeset created a local variable which the second typeset made readonly, so that the final assignment caused an error. Then the `$msg` in the function went out of scope, and the ordinary parameter, with no readonly restriction, was visible again.

I mentioned another special typeset option in the previous chapter:

```
typeset -T TEXINPUTS texinputs
```

to tie together the scalar `$TEXINPUTS` and the array `$texinputs` in the same way that `$PATH` and `$path` work. This is a one-off; it's the only time `typeset` takes exactly two parameter names on the command line. All other uses of typeset take a list of parameters to which any flags given are applied. See the manual for the remaining flags, although most of the more interesting ones have been discussed.

The other thing you need to know about flags is that you use them with a '`+`' sign to turn off the corresponding attribute. So

```
typeset +r msg
```

allows you to set `$msg` again. From version `4.1`, you won't be able to turn off the readonly attribute for a special parameter; that's because there's too much scope for confusion, including attempting to set constant strings in the code. For example, '`$ZSH_VERSION`' always prints a fixed string; attempting to change that is futile.

The final use of typeset is to list parameters. If you type '`typeset`' on its own, you get a complete list of parameters and their values. From 3.1.7, you can turn on the flag `-H` for a parameter, which means to hide its value while you're doing this. This can be useful for some

of the more enormous parameters, particularly special parameters which I'll talk about in the section in chapter 7 on modules, which tend to swamp the display `typeset` produces.

You can also list parameters of a particular type, by listing the flags you want to know about. For example,

```
typeset -r
```

lists all readonly parameters. You might expect '`typeset +r`' to list parameters which *don't* have that attribute, but actually it lists the same parameters but without showing their value. '`typeset +`' lists all parameters in this way.

Another good way of finding out about parameters is to use the special expansion '`${(t)`*param*`}`', for example

```
print ${(t)PATH}
```

prints '`scalar-export-special`': `$PATH` is a scalar parameter, with the `-x` flag set, and has a special meaning to the shell. Actually, '`special`' means something a bit more than that: it means the internal code to get and set the parameter behaves in a way which has side effects, either to the parameter itself or elsewhere in the shell. There are other parameters, like `$HISTFILE`, which are used by the shell, but which are get and set in a normal way — they are only special in that the value is looked at by the shell; and, after all, any old shell function can do that, too. Contrast this with `$PATH` which has all that paraphernalia to do with hashing commands to take care of when it's set, as I discussed above, and I hope you'll see the difference.

Reading into parameters

The '`read`' builtin, as its name suggests, is the opposite to '`print`' (there's no '`write`' command in the shell, though there is often an external command of that name to send a message to another user), but reading, unlike printing, requires something in the shell to change to take the value, so unlike `print`, `read` is forced to be a builtin. Inevitably, the values are read into a parameter. Normally they are taken from standard input, very often the terminal (even if you're running a script, unless you redirected the input). So the simplest case is just

```
read param
```

and if you type a line, and hit return, it will be put into `$param`, without the final newline.

The `read` builtin actually does a bit of processing on the input. It will usually strip any initial or final whitespace (spaces or tabs) from the line read in, though any in the middle are kept. You can read a set of values separated by whitespace just by listing the parameters to assign them to; the last parameter gets all the remainder of the line without it being split. Very often it's easiest just to read into an array:

```
% read -A array
      this is a line typed in now, \
    by me,    in this   space
% print ${array[1]} ${array[12]}
this space
```

(I'm assuming you're using the native zsh array format, rather than the one set with KSH_ARRAYS, and shall continue to assume this.)

It's useful to be able to print a prompt when you want to read something. You can do this with 'print -n', but there's a shorthand:

```
% read line'?Please enter a line: '
Please enter a line: some words
% print $line
some words
```

Note the quotes surround the '?' to prevent it being taken as part of a pattern on the command line. You can quote the whole expression from the beginning of 'line', if you like; I just write it like that because I know parameter names don't need quoting, because they can't have funny characters in. It's almost logical.

Another useful trick with read is to read a single character; the '-k' option does this, and in fact you can stick a number immediately after the 'k' which specifies a number to read. Even easier, the '-q' option reads a single character and returns status 0 if it was y or Y, and status 1 otherwise; thus you can read the answer to yes/no questions without using a parameter at all. Note, however, that if you don't supply a parameter, the reply gets assigned in any case to $REPLY if it's a scalar — as it is with -q — or $reply if it's an array — i.e. if you specify -A, but no parameter name. These are more examples of the non-special parameters which the shell uses — it sets $REPLY or $reply, but only in the same way you would set them; there are no side-effects.

Like print, read has a -r flag for raw mode. However, this just has one effect for read: without it, a \ at the end of the line specifies that the next line is a continuation of the current one (you can do this when you're typing at the terminal). With it, \ is not treated specially.

Finally, a more sophisticated note about word-splitting. I said that, when you are reading to many parameters or an array, the word is split on whitespace. In fact the shell splits words on any of the characters found in the (genuinely special, because it affects the shell's guts) parameter $IFS, which stands for 'input field separator'. By default — and in the vast majority of uses — it contains space, tab, newline and a null character (character zero: if you know that these are usually used to mark the end of strings, you might be surprised the shell handles these as ordinary characters, but it does, although printing them out usually doesn't show anything). However, you can set it to any string: enter

```
fn() {
  local IFS=:
  read -A array
  print -l $array
}
fn
```

and type

```
one word:two words:three words:four
```

The shell will show you what's in the array it's read, one 'word' per line:

```
one word
```

```
two words
three words
four
```

You'll see the bananas, er, words (joke for the over-thirties) have been treated as separated by a colon, not by whitespace. Making `$IFS` local didn't work in old versions of zsh, as with other specials; you had to save it and restore it.

The `read` command in zsh doesn't let you do line editing, which some shells do. For that, you should use the `vared` command, which runs the line editor to edit a parameter, with the `-c` option, which allows `vared` to create a new parameter. It also takes the option `-p` to specify a prompt, so one of the examples above can be rewritten

```
vared -c -p 'Please enter a line: ' line
```

which works rather like read but with full editing support. If you give the option `-h` (history), you can even retrieve values from previous command lines. It doesn't have all the formatting options of read, however, although when reading an array (use the option `-a` with `-c` if creating a new array) it will perform splitting.

Other builtins to control parameters

The remaining builtins which handle parameters can be dealt with more swiftly.

The builtin `set` simply sets the special parameter which is passed as an argument to functions or scripts, and which you access as `$*` or `$@`, or `$<number>` (Bourne-like format), or via `$argv` (csh-like format), known however you set them as the 'positional parameters':

```
% set a whole load of words
% print $1
a
% print $*
a whole load of words
% print $argv[2,-2]
whole load of
```

It's exactly as if you were in a function and had called the function with the arguments '`a whole load of words`'. Actually, set can also be used to set shell options, either as flags, e.g. '`set -x`', or as words after '`-o`' , e.g. '`set -o xtrace`' does the same as the previous example. It's generally easier to use `setopt`, and the upshot is that you need to be careful when setting arguments this way in case they begin with a '-'. Putting '`--`' before the real arguments fixes this.

One other use of `set` is to set any array, via

```
set -A any_array words to assign to any_array
```

which is equivalent to (and the standard Korn shell version of)

```
any_array=(words to assign to any_array)
```

One case where the `set` version is more useful is if the name of an array itself comes from a parameter:

```
arrname=myarray
set -A $arrname words to assign
```

has no easy equivalent in the other form; the left hand side of an ordinary assignment won't expand a parameter:

```
# Doesn't work; syntax error
$arrname=(words to assign)
```

This worked in old versions of zsh, but that was on the non-standard side. The `eval` command, described below, gives another way around this.

Next comes '`shift`', which simply moves an array up one element, deleting the original first one. Without an array name, it operates on the positional parameters. You can also give it a number to shift other than one, before the array name.

```
shift array
```

is equivalent to

```
array=(${array[2,-1]})
```

(almost — I'll leave the subtleties here for the chapter on expansion) which picks the second to last elements of the array and assigns them back to the original array. Note, yet again, that `shift` operates using the *name*, not the *value* of the array, so no '`$`' should appear in front, otherwise you get something similar to the trick I showed for '`set -A`'.

Finally, `unset` unsets a parameter, and I already showed you could unset a key/value pair of an associative array. There is one subtlety to be mentioned here. Normally, `unset` just makes the parameter named disappear off the face of the earth. However, if you call `unset` in a function, its ghost lives on in the sense that any parameter you create in the same name will be scoped as the original parameter was. Hence:

```
var='global value'
fn() {
  typeset var='local value'
  unset var
  var='what about this?'
}
fn
print $var
```

The final statement prints '`global value`': even though the local copy of `$var` was unset, the shell remembers that it was local, so the second `$var` in the function is also local and its value disappears at the end of the function.

3.2.7 History control commands

The easiest way to access the shell's command history is by editing it directly. The second easiest way is to use the '`!`'-history mechanism. Other ways of manipulating it are based around the `fc` builtin, which probably once stood for something (according to Oliver Kiddle, 'fix command', which is as good as anything). I talked quite a bit about it in the last chapter, and don't really have anything to add. Just note that the two other commands based around it are `history` and `r`.

3.2.8 Job control and process control

One of the major contributions of the C-shell was job control. You need to know about foreground and background tasks, and again I introduced these in the last chapter along with the options that control them. Here is an introduction to the relevant builtins.

You start a background job in two ways. First, directly, by putting an '`&`' after it:

```
sleep 10 &
```

and secondly by starting it in the normal way (i.e. in the foreground), then typing `^Z`, and using the `bg` command to put it in the background. Between typing `^Z` and `bg`, the job is still there, but is not running; it is 'suspended' or 'stopped' (systems use different descriptions for the same thing), waiting for you to decide what to do with it. In either case, the job then continues without the shell waiting for it. It will still try and read from or write to the terminal if that's how you started it; you need to use the shell's redirection facilities right at the start if you want to change that, there's nothing you can do after the job has already started.

By the way, 'sleep' isn't a builtin. Oddly enough, you can suspend a builtin command or sequence of commands (such as shell function) with `^Z`, although since the shell has to continue executing your commands as well as being suspended, it does the only thing it can do — fork, so that the commands you suspend are put into the background. Probably you will only rarely do this with builtins. No other shell, so far as I know, has this feature.

A job will stop if it needs to read from the terminal. You see a message like:

```
[1]  + 1348 suspended (tty input)  jobname and arguments
```

which means the job is suspended very much like you had just typed `^Z`. You need to bring the job into the forground, as described below, so that you can type something to it.

By the way, the key to type to suspend a command may not be `^Z`; it usually is, but that can be changed. Run '`stty -a`' and look for what is listed after '`susp =`' — probably, but not necessarily, `^Z`. So if you want to use another character — it must be a single character; this is handled deep in the terminal interface, not in the shell — you can run

```
stty susp '^]'
```

or whatever. You will note from the `stty` output that various other job control characters can be changed similarly. The `stty` command is external and its format for both output and input can vary quite a bit from system to system.

Instead of putting the command into the background, you can bring it back to the foreground again with `fg`. This is useful for temporarily stopping what you are doing so you can do something else. These days you would probably do it in another window; in the old days when people logged in from simple terminals this was even more useful. A typical example of this is

```
more file                          # look at file
^Z                                 # suspend
[1] + 8592 suspended  more file    # message printed
...                                # do something else
fg %1                              # resume the 'more'
```

The '`%`' is the usual way of referring to jobs. The number after it is what appeared in square brackets with the suspended message; I don't know why the shell doesn't use the '`%`' notation there, too. You also see that with the 'continued' message when you put something into the background, and again at the end with the 'done' message which tells you a background job is finished. The '`%`' can take other forms; the most common is to follow it by the name of a command, such as '`%more`' in this case. The forms `%+` and `%-` refer to the most recent and second most recent jobs — the '`+`' in the 'suspended' message is telling you that the `more` job could be referred to like that.

Most of the job control commands will actually assume you are talking about '`%+`' if you don't give an argument, so assuming I hadn't started any other commands in the background, I could just have put '`fg`' at the end of the sequence of commands above. This actually cuts both ways: `fg` is the default operation on jobs referred to with the '`%`' notation, so just typing '`%1`' with no command name would have worked, too.

You can jog your memory about what's going on with the '`jobs`' command. It looks like a series of messages of the form beginning with the number in square brackets; usually the jobs will either be 'running' or 'suspended'. This will tell you the numbers you need.

One other useful thing you can do with a job is to tell the shell to forget about it. This is only really useful if it is already running in the background; then you can run '`disown`' with the job identifier. It's useful for jobs you want to continue after you've logged out, as well as jobs that have their own windows which you can therefore control directly. With disowned jobs, the shell doesn't warn you that they are still there when you log out. You can actually disown a background job when you start it by putting '`&|`' or '`&!`' at the end of the line instead of simply '`&`'. Note that if the job was suspended when you disowned it, it will stay disowned; this is pretty pointless, so you probably should run '`bg`' on it first.

The next most likely thing you want to do with a job is kill it, or maybe suspend it when it's already in the background and you can't just type `^Z`. This is where the `kill` builtin comes in. There's more to this than there is to the builtins mentioned above. First, you can use `kill` with other processes that weren't started from the current shell. In that case, you would use a number to identify it, with no `%` — that's why the `%`'s were there in the other cases. Of course, you need to find out the number; the usual way is with the `ps` command, which is not a builtin but which appears on all UNIX-like systems. As a stupid example, here I start a disowned process which does very little, look for it, then kill it:

```
% sleep 60 &|
% ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
pws        623   614  0 22:12 pts/0    00:00:00 zsh
pws       8613   623  0 23:12 pts/0    00:00:00 sleep 60
```

```
pws       8615   623  0 23:12 pts/0    00:00:00 ps -f
% kill 8613
% ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
pws        623   614  0 22:12 pts/0    00:00:00 zsh
pws       8616   623  0 23:12 pts/0    00:00:00 ps -f
```

The process has disappeared the second time I look. Notice that in the usual lugubrious UNIX way the shell didn't bother to tell you the process had been killed; however, it will report an error if it failed to send it the signal. Sending it the signal is all the shell cares about; the shell won't warn if you if the process decided it didn't want to die when told to, so it's still a good idea to check.

Sometimes you want to wait for a process to exit; the `wait` builtin can do this, and like `kill` can take a process number as well as a job number. However, that's a bit deceptive — you can't actually wait for a process which wasn't started directly from the shell. Indeed, the mechanism for waiting is all bound up with the way UNIX handles processes; unless its parent waits for it, a process becomes a 'zombie' and hangs around until the system's foster parent, the 'init' process (always process number 1) waits for it instead. It's all a little bit baroque, but for the shell user, wait just means you can hang on until something you started has finished. Indeed, that's how foreground processes work: the shell in effect uses the internal version of `wait` to hang around until the job exits. (Well, actually that's a lie; the system wakes it up from whatever it's doing to tell it a child has finished, so all it has to do is doze off to wait.)

Furthermore, you can wait for a process even if job control isn't running. Job control, basically anything involving those `%`'s, is only useful when you are sitting at a terminal fiddling with commands; it doesn't operate when you run scripts, say. Then the shell has much less freedom in how to control its jobs, but it can still wait for a background process, and it can still use `kill` on a process if it knows its number. For this purpose, the shell stores the ID of the last process started in the background in the parameter `$!`; there's probably a good reason for the '`!`', but I don't know what it is. This happens regardless of job control.

Signals

The `kill` command can do a good deal more than just kill a process. That is the default action, which is why the command has that name. But what it's really doing is sending a 'signal' to a process. Signals are the simplest way of communicating to another process; in fact, they are about the only simple way if you haven't made special arrangements for the process to read messages from you. Signal names are written like `SIGINT`, `SIGTSTP`, `SIGKILL`; to send a particular signal to a process, you remove the `SIG`, stick a hyphen in front, and use that as the first argument to `kill`, e.g.:

```
kill -KILL 8613
```

Some of the things you already know about are actually doing just that. When you type `^C` to stop a process, you are actually sending it a `SIGINT` for 'interrupt', as if you had done

```
kill -INT 8613
```

The usual signal sent by `kill` is not, as you might have guessed, `SIGKILL`, but actually `SIGTERM` for 'terminate'; `SIGKILL` is stronger as the process can't block that signal, as it can

with many (we'll see how the shell can do that in a moment). It's familiar to UNIX hackers as '`kill -9`', because all the signals also have numbers. You can see the list of signals in zsh by doing:

```
% print $signals
EXIT HUP INT QUIT ILL TRAP ABRT BUS FPE KILL USR1
SEGV USR2 PIPE ALRM TERM STKFLT CLD CONT STOP TSTP
TTIN TTOU URG XCPU XFSZ VTALRM PROF WINCH POLL PWR
UNUSED ZERR DEBUG
```

Your list will probably be different from mine; this is for Linux, and the list is very system-specific, even though the first nine are generally the same, and many of the others are virtually always present. Actually, `SIGEXIT` is an invention by the shell for you to allow the shell to do something when a function exits (see the section on 'traps' below); you can't actually use '`kill -EXIT`'. Thus `SIGHUP` is the first real signal, and indeed that's number one, so you have to shift the contents of `$signals` along one to get the right numbers. `SIGTERM` and `SIGINT` usually have the same effect, stopping the process, unless that has decided to handle the signal some other way.

The last two signals are bogus, too: `SIGZERR` is to allow the shell to do something on an error (non-zero exit status), while with `SIGDEBUG` you can do it on every command. Again, the 'something' to be executed is a 'trap', as I'll discuss in a short while.

Typing `^Z` to suspend a process actually sends the process a `SIGTSTP` (terminal stop, since it usually comes from the terminal), while `SIGSTOP` is similar but usually doesn't come from a terminal. Even restarting a process as with `bg` sends it a signal, in this case `SIGCONT`. It seems a bit odd to signal a process to restart; why can't the operating system just restart it when you ask? The real answer is probably that signals provide an easy way for you to talk to the operating system without grovelling around in the dirt too much.

Before I talk about how you make the shell handle signals it receives, there is one extra oddment: the `suspend` builtin effectively sends the shell a signal to suspend it, as if you'd typed `^Z`, though as you've probably found by now that doesn't suspend the shell itself. It's only useful to do this if the shell is running under some other programme, else there's no way of restoring it and suspending is effectively the same as exiting the shell. For this reason, the shell won't let you call `suspend` in a login shell, because it assumes that is running as the top level (though in the previous chapter you learnt there's actually nothing that special about login shells; you can start one just with 'zsh -l'). If you're logged in remotely via `rsh` or `ssh`, it's usually more convenient to use the keystrokes '`~^Z`' which those define, rather than zsh's mechanism; they have to be at the beginning of a line, so hit return first if necessary. This returns you to your local terminal; you can resume the remote login with '`fg`' just like any other programme.

Traps

The way of making the shell handle signals is called 'traps'. There are actually two mechanisms for this. I'll present the more standard one and then talk about the advantages and drawbacks of the other one at the end.

The standard version (shared with other shells) is via the '`trap`' builtin. The first argument is a chunk of shell code to execute, which obviously needs to be quoted when you pass it as an argument, and the remaining arguments are a list of signals to handle, minus the `SIG` prefix. So:

```
trap "echo I\\'m trapped." INT
```

tells the shell what to do on `SIGINT`, i.e. `^C`. Note the extra layer of quoting: the double quotes surround the code, so that when they are stripped `trap` sees the chunk

```
echo I\'m trapped
```

Usually the shell would abort what it was doing and return to the main prompt when you hit `^C`. Now, however, it will simply print the message and carry on. You can try this, for example, with

```
read line
```

If you hit `^C` while it's waiting for input, you'll see the message go up, but the shell will still wait for you to type a line.

A warning about this: `^C` is only trapped within the shell itself. If you start up an external programme, it will have its own mechanism for handling signals, and if it usually aborts on `^C` it still will. But there's a sting in the tail: do

```
cat
```

which waits for input to output again (you need to use `^D` to exit normally). If you type `^C` here, the command will be aborted, as I said — but you still get the message '`I'm trapped`'. That's because the shell is able to tell that the command got that particular signal, and calls the trap when the `cat` exits. Not all shells do this; furthermore, some commands which handle signals themselves won't give the shell enough information to know that a signal arrived, and in that case the trap won't be called. Such commands are usually the more sophisticated things like editors or screen managers or whatever; you just have to find out by trial and error.

You can also make the shell ignore the signal completely. To do this, the first argument should be an empty string:

```
trap '' INT
```

Now `^C` will have no effect, and *this* time the effect *is* passed on directly to commands called from the shell — try the `cat` example and you won't be able to interrupt it; type `^D` or use the lesser known but more powerful `^\` (control with backslash), which sends `SIGQUIT`. If it hasn't been disabled, this will also produce a file `core`, which contains debugging information about what the programme was doing when it exited — never call your own files `core`. You can trap `SIGQUIT` too, if you want. (The shell itself usually ignores `SIGQUIT`; it's only useful for external commands.)

Now the other sort of trap. I could have written for the first example:

```
TRAPINT() {
  print I\'m trapped.
}
```

As you can see, this is just a function: functions beginning `TRAP` are special. However, it's a real function too; you can call it by hand with the command 'TRAPINT', and it will run perfectly happily with no funny side effects.

There is a difference between the way the two types work. In the 'trap' sort of trap, the code is just evaluated just as if it appeared as instructions to the shell at the point where the trap happened. So if you were in a function, you would see the environment of that function with its local variables; if you set a local variable with typeset, it would be visible in the function just as if it were created there.

However, in the function type of trap, the code is provided with its own function environment. Now if you use typeset the parameter created is local only to the trap. In most cases, that's all the difference there is; it's up to you to decide which is more convenient. As you can see, the function type of trap doesn't require the extra layer of quoting, so looks a little smarter. Conveniently, the 'trap' command on its own lists all traps in the form of the shell code you'd need to recreate them, and you can see which sort is which.

There are two cases where the difference sticks out. One is that the function type has some extra wiring to allow you both to trap a signal, and pretend to anyone watching that the shell didn't handle it. An example will show this:

```
TRAPINT() {
  print "Signal caught, stopping anyway."
  return $(( 128 + $1 ))
}
```

That second line may look as rococo as the Amalienburg, but it's meaning is this: $1, the first argument to the function, is set to the number of the signal. In this case it will be 2 because that's the standard number for SIGINT. That means the arithmetic substitution $((...)) returns 130, the command 'return 130' is executed, and the function returns with status 130. Returning with non-zero status is special in function traps: it tells the shell you want to abort the surrounding command even though the trap was handled, and that you want the status associated with that to be 130. It so happens that this is how UNIX handles returns from normal traps. Without setting a trap, do

```
% cat
^C
% print $?
```

and you'll see that this, too, has given the status 130, 128 plus the value of SIGINT. So if you *do* have the trap set, you'll see the message, but the command will abort — even if it was running inside the shell.

Try

```
% read line
^C
```

to see that happening. If you look at the status in $? you'll find it's actually 1, not 130; that's because the read command, when it aborted, overrode the return value from the trap. But it does that with an untrapped ^C, too, so that's not really an exception to what I've just said.

If you've been paying attention, you'll realise that traps set with the trap builtin can't do it in quite this way, because the function they return from would be whatever function you were in. You can see that:

```
trap 'echo Returning...; return;' INT
```

```
fn() {
  print In fn...
  read param
  print Leaving fn..
}
```

If you run `fn` and hit `^C`, the signal is trapped and the message printed, but because of the `return`, the shell quits `fn` immediately and you don't see the final message. If you missed out the '`return;`' (try it), the shell would carry on with the rest of `fn` after you typed something to `read`. Of course you can use this mechanism to leave functions after trapping a signal; it just so happens that in this case the mechanism with `TRAPINT` is a little closer to what untrapped signals do and hence a little neater.

One final flourish of late Baroque splendour: the trap for `SIGEXIT`, the one called when a function (or the shell itself, in fact) exits is a bit special because in the case of exiting a function it will be called in the environment of the calling function. So if you need to do something like set a local variable for an enclosing function you can have

```
trap 'typeset param_in_enclosing_func=value' EXIT
```

do it for you; you couldn't do that with `TRAPEXIT` because the code would have its own function, so that even though it would be called after the first function exited, it wouldn't run directly in the enclosing one but in a separate `TRAPEXIT` function. You can even set an EXIT trap for the enclosing function by defining a nested '`trap .. EXIT`' inside that trap itself.

I lied, because there is one more special thing about `TRAPEXIT`: it's always reset after you exit a function and the trap itself has been called. Most traps just hang around until you explicitly unset them. There is an option, `LOCAL_TRAPS`, which makes traps set inside functions as well insulated as possible from those outside, or inside deeper functions. In other words, the old trap is saved and then restored when you exit the function; the scoping works pretty much like that for `typeset`, and in the same way traps for the enclosing scope, apart from any for `EXIT`, remain in effect inside a function unless you explicitly override them; and, again in the same way, if you unset it inside the function it will still be restored on exit.

`LOCAL_TRAPS` is the fixed behaviour of some other shells. In zsh, without the option set:

```
trap 'echo Hi.' INT
fn() {
   trap 'echo Bye.' INT
}
```

Calling `fn` simply replaces the trap defined outside the function with the one defined inside while:

```
trap 'echo Hi.' INT
fn() {
   setopt localtraps
   trap 'echo Bye.' INT
}
```

puts the original 'Hi' trap back after the function exits.

I haven't told you how to unset a trap for good: the answer is

```
trap - INT
```

As you would guess, you can use `unfunction` with function-type traps; that will correctly remove the trap as well as deleting the function. However, '`trap -`' works with both, so that's the recommended way.

Limits on processes

One other way that jobs started by the shell can be controlled is by using limits. These are actually limits set by the operating system, but the shell gives you a way of controlling them: the `limit` and `unlimit` commands. Type '`limit`' on its own to see a summary. I get:

```
cputime         unlimited
filesize        unlimited
datasize        unlimited
stacksize       8MB
coredumpsize    0kB
memoryuse       unlimited
maxproc         2048
descriptors     1024
memorylocked    unlimited
addressspace    unlimited
```

where the item on the left of each line is what is being limited, and on the right is the value. The manual page to look at, at least on Linux is for the function `getrusage`; that's the function the shell is calling when you run `limit` or `unlimit`.

In this case, the items are:

`cputime` the total CPU time used by a process

`filesize` maximum size of a file

`datasize` the maximum size of data in use by a programme

`stacksize` the maximum size of the stack, which is the area of memory used to store information during function calls

`coredumpsize` the maximum size of a `core` file, which is an image of memory left by a programme that crashes, allowing you to debug it with `gdb`, `dbx`, `ddd` or some other debugger

`memoryuse` the maximum main memory, i.e. programme memory which is in active use and hasn't been 'swapped out' to disk

`maxproc` the maximum number of simultaneous processes

`descriptors` the maximum number of simultaneously open files ('descriptors' are the internal mechanism for referring to an open file on UNIX-like systems)

`memorylocked` the maximum amount of memory locked in (I don't know what that is, either)

`addressspace` the total amount of virtual memory, i.e. any memory whether it is main memory, or refers to somewhere on a disk, or indeed anything else.

You may well see other names; the shell decides when it is compiled what limits are supported by the system.

Of those, the one I use most commonly is `coredumpsize`: sometimes when I'm debugging a file I want a crashed programme to produce a '`core`' files so I can run `gdb` or `dbx` on it ('`unlimit coredumpsize`'), while other times they are just untidy ('`limit coredumpsize 0`'). Probably you would only alter any of the others if you knew there was a problem, for example a number-crunching programme used so much memory that the rest of the system was badly affected and you wanted to limit `datasize` to 64 megabyte or whatever. You could write this as:

```
limit datasize 64m
```

There is a distinction made between 'hard' and 'soft' limits. Both have the same effect on programmes, but you can remove or reduce 'soft' limits, while only the superuser (the system administrator's login, root) can do that to 'hard' limits. Usually, therefore, `limit` and `unlimit` manipulate soft limits; to show or set hard limits, give the option `-h`. If I do '`limit -h`', I get the same list of limits as above, but with `stacksize` and `coredumpsize` unlimited — that means I can reduce or remove the limits on those if I want, they're just set for my own convenience.

Why is `stacksize` set in this way? As I said, it refers to the memory in which the functions in programmes store variables and any other local information. If one function calls another, it uses more memory. You can get into a situation where functions call themselves recursively and there is no way out until the machine runs out of memory; limiting `stacksize` prevents this. You can actually see this with zsh itself (probably better not to try this if you'd rather the shell you're running didn't crash):

```
% fn() { fn; }
% fn
```

defines a function which keeps calling itself. To do this, all the functions *inside* zsh are calling themselves as well, using more and more stack memory. Actually, zsh uses other forms of memory inside each function and my version of zsh crashes due to exhaustion of that memory instead. However, it depends on the system how this works out.

Times

One way of returning information on process resources is with the '`times`' command. It simply shows the total CPU time used by the shell and by the programmes called for it — in that order, and without description, so you need to remember. On each line, the first number is the time spent in user space and the second is the time spent in system space. If you're not concerned about the details of programmes the difference is pretty irrelevant, but if you are, then the difference is very roughly that between the time spent in the code you actually see before you compile a programme, and the time spent in 'hidden' code where the system is doing something for you. It's not such an obvious distinction, because many library routines, such as mathematical functions, are run in user mode as no privileged access to internal bits of the system is required. Typically, system time is concerned with the details of input and output — though even there it's not so simple, because the C output routines `printf`, `puts`, `fread` and others have user mode code which then calls the system routines `read`, `write` and so on.

You can measure the time taken by a particular external command by putting 'time', in the singular this time, in front of it; this is essentially another precommand modifier, and is a shell reserved word rather than a builtin. This gives fairly obvious information. You can specify the information using the $TIMEFMT parameter, which has its own percent escapes, different from the ones used in prompts. It exists partly because the shell allowed you to access all sorts of other information about the process which ran, such as 'page faults' — occasions when the system had to fetch a part of the programme or data from disk because it wasn't in the main memory. However, that disappeared because it was too much work to convert the feature to configure itself automatically for different operating systems. It may be time to resurrect it.

You can also force the time to be shown automatically by setting the parameter $REPORTTIME; if a command runs for more than this many seconds, the $TIMEFMT output will be shown automatically.

3.2.9 Terminals, users, etc.

Watching for other users

Although this is more associated with parameters than builtins, the 'log' command will tell you whether any of a group of people you want to watch out for have logged in or out. To use this, you set the $watch array parameter to a list of user names, or 'all' for everyone, or 'notme' for everyone except yourself. Even if you don't use log, any changes will be reported just before the shell prints a prompt. It will be printed using the $WATCHFMT parameter: once again, this takes its own set of percent escapes, listed in the zshparam manual.

ttyctl

There is a command ttyctl which is designed to keep badly behaved external commands from messing up the terminal settings. Most programmes are careful to restore any settings they change, but there are exceptions. After 'ttyctl -f', the terminal is frozen; zsh will restore the settings, no matter what an external programme does with it. This includes deliberate attempts to change the terminal settings with the 'stty' command, so the default is unfrozen, 'ttyctl -u'.

3.2.10 Syntactic oddments

This section collects together a few builtins which, rather than controlling the behaviour of some feature of the shell, have some other special effect.

Controlling programme flow

The four functions here are exit, return, break, continue and source or .: they determine what the shell does next. You've met exit — leave the shell altogether — and return — leave the current function. Be very careful not to confuse them. Calling exit in a shell function is usually bad:

```
% fn() { exit; }
```

```
% fn
```

This makes you entire shell session go away, not just the function. If you write C programmes, you should be very familiar with both, although there is one difference in this case: `return` at the top level in an interactive shell actually does nothing, rather than leaving the shell as you might expect. However, in a script, return outside a function *does* cause the entire script to stop. The reason for this is that zsh allows you to write autoloaded functions in the same form as scripts, so that they can be used as either; this wouldn't work if `return` did nothing when the file was run as a script. Other shells don't do this: `return` does nothing at the top level of a script, as well as interactively. However, other shells don't have the feature that function definition files can be run as scripts, either.

The next two commands, `break` and `continue`, are to do with constructs like '`if`'-blocks and loops, and it will be much easier if I introduce them when I talk about those below. They will also already be familiar to C programmers. (If you are a FORTRAN programmer, however, `continue` is *not* the statement you are familiar with; it is instead equivalent to `CYCLE` in FORTRAN90.)

The final pair of commands are `.` and `source`. They are similar to one another and cause another file to be read as a stream of commands in the current shell — not as a script, for which a new shell would be started which would finish at the end of the script. The two are intended for running a series of commands which have some effect on the current shell, exactly like the startup files. Indeed, it's a very common use to have a call to one or other in a startup file; I have in my `~/.zshrc`

```
[[ -f ~/.aliasrc ]] && . ~/.aliasrc
```

which tests if the file `~/.aliasrc` exists, and if so runs the commands in it; they are treated exactly as if they had appeared directly at that point in `.zshrc`.

Note that your `$path` is used to find the file to read from; this is a little surprising if you think of this as like a script being run, since zsh doesn't search for a script, it uses the name exactly as you gave it. In particular, if you don't have '`.`' in your `$path` and you use the form '`.`' rather than '`source`' you will need to say explicitly when you want to source a file in the current directory:

```
. ./file
```

otherwise it won't be found.

It's a little bit like running a function, with the file as the function body. Indeed, the shell will set the positional parameters `$*` in just the same way. However, there's a crucial difference: there is no local parameter scope. Any variables in a sourced file, as in one of the startup files, are in the same scope as the point from which it was started. You can, therefore, source a file from inside a function and have the parameters in the sourced file local, but normally the only way of having parameters only for use in a sourced file is to unset them when you are finished.

The fact that both `.` and `source` exist is historical: the former comes from the Bourne shell, and the latter from the C shell, which seems deliberately to have done everything differently. The point noted above, that source always searches the current directory (and searches it first), is the only difference.

Re-evaluating an expression

Sometimes it's very useful to take a string and run it as if it were a set of shell commands. This is what `eval` does. More precisely, it sticks the arguments together with spaces and calls them. In the case of something like

```
eval print Hello.
```

this isn't very useful; that's no different from a simple

```
print Hello.
```

The difference comes when what's on the command line has something to be expanded, like a parameter:

```
param='print Hello.'
eval $param
```

Here, the `$param` is expanded just as it would be for a normal command. Then `eval` gets the string '`print Hello.`' and executes it as shell command line. Everything — really everything — that the shell would normally do to execute a command line is done again; in effect, it's run as a little function, except that no local context for parameters is created. If this sounds familiar, that's because it's exactly the way traps defined in the form

```
trap 'print Hello.' EXIT
```

are called. This is one simple way out of the hole you can sometimes get yourself into when you have a parameter which contains the name of another parameter, instead of some data, and you want to get your hands on the data:

```
# somewhere above...
origdata='I am data.'
# but all you know about is
paramname=origdata
# so to extract the data you can do...
eval data=\$$paramname
```

Now `$data` contains the value you want. Make sure you understand the series of expansions going on: this sort of thing can get very confusing. First the command line is expanded just as normal. This turns the argument to `eval` into '`data=$origdata`'. The '`$`' that's still there was quoted by a backslash; the backslash was stripped and the '`$`' left; the `$paramname` was evaluated completely separately — quoted characters like the `\$` don't have any effect on expansions — to give `origdata`. Eval calls the new line '`data=$origdata`' as a command in its own right, with the now obvious effect. If you're even slightly confused, the best thing to do is simply to quote everything you don't want to be immediately expanded:

```
eval 'data=$'$paramname
```

or even

```
eval 'data=${'$paramname'}'
```

may perhaps make your intentions more obvious.

It's possible when you're starting out to confuse '`eval`' with the `` `...` `` and `$(...)` commands, which also take the command in the middle '. . .' and evaluate it as a command line. However, these two (they're identical except for the syntax) then insert the output of that command back into the command line, while `eval` does no such thing; it has no effect at all on where input and output go. Conversely, the two forms of command substitution don't do an extra level of expansion. Compare:

```
% foo='print bar'
% eval $foo
bar
```

with

```
% foo='print bar'
% echo $($foo)
zsh: command not found: print bar
```

The `$(...)` substitution took `$foo` as the command line. As you are now painfully aware, zsh doesn't split scalar parameters, so this was turned into the single word '`print bar`', which isn't a command. The blank line is '`echo`' printing the empty result of the failed substitution.

3.2.11 More precommand modifiers: `exec`, `noglob`

Sometimes you want to run a command *instead* of the shell. This sometimes happens when you write a shell script to process the arguments to an external command, or set parameters for it, then call that command. For example:

```
export MOZILLA_HOME=/usr/local/netscape
netscape "$@"
```

Run as a script, this sets an environment variable, then starts `netscape`. However, as always the shell waits for the command to finish. That's rather wasteful here, since there's nothing more for the shell to do; you'd rather it simply magically turned into the `netscape` command. You can actually do this:

```
export MOZILLA_HOME=/usr/local/netscape
exec netscape "$@"
```

'`exec`' tells the shell that it doesn't need to wait; it can just make the command to run replace the shell. So this only uses a single process.

Normally, you should be careful not to use `exec` interactively, since normally you don't want the shell to go away. One legitimate use is to replace the current zsh with a brand new one if (say) you've set a whole load of options you don't like and want to restore the ones you usually have on startup:

```
exec zsh
```

Or you may have the bad taste to start a completely different shell altogether. Conversely, a good piece of news about `exec` is that it is common to all shells, so you can use it from another shell to start zsh in the way I've just shown.

Like '`command`' and '`builtin`', '`exec`' is a 'precommand modifier' in that it alters the way a command line is interpreted. Here's one more:

```
noglob print *
```

If you've remembered what 'glob' means, this is all fairly obvious. It instructs the shell not to turn the '`*`' into a list of all the files in the directory, but instead to let well alone. You can do this by quoting the '`*`', of course; often `noglob` is used as part of an alias to set up commands where you never need filename generation and don't want to have to bother quoting everything. However, note that `noglob` has no effect on any other type of expansion: parameter expansion and backquote (`` `...` ``) expansion, for example, happen as normal; the only thing that doesn't is turning patterns into a list of matching files. So it doesn't take away the necessity of knowing the rules of shell expansion. If you need that, the best thing to do is to use `read` or `vared` (see below) to read a line into a parameter, which you pass to your function:

```
read -r param
print $param
```

The `-r` makes sure `$param` is the unadulterated input.

3.2.12 Testing things

I told you in the last chapter that the right way to write tests in zsh was using the '`[[ ... ]]`' form, and why. So you can ignore the two builtins '`test`' and '`[`', even though they're the ones that resemble the Bourne shell. You can safely write

```
if [[ $foo = '' ]]; then
  print The parameter foo is empty.  O, misery me.
fi
```

or

```
if [[ -z $foo ]]; then
  print Alack and alas, foo still has nothing in it.
fi
```

instead of monstrosities like

```
if test x$foo != x; then
  echo The emptiness of foo.  Yet are we not all empty\?
fi
```

because even if `$foo` does expand to an empty string, which is what is implied if the tests are true, '`[[ ... ]]`' remembers there was something there and gets the syntax right. Rather than a builtin, this is actually a reserved word — in fact it has to be, to be syntactically special — but you probably aren't too bothered about the difference.

There are two sorts of tests, both shown above: those with three arguments, and those with two. The three-argument forms all have some comparison in the middle; in addition to '=' (or '==', which means the same here, and which according to the manual page we should be using, though none of us does), there are '`!=`' (not equal), '<', '>', '<=' and '>='. All these do *string* comparisons, i.e. they compare the sort order of the strings.

Since there are better ways of sorting things in zsh, the '=' and '`!=`' forms are by far the most common. Actually, they do something a bit more than string comparison: the expression on the right can be a pattern. The patterns understood are just the same as for matching filenames, except that '/' isn't special, so it can be matched by a '`*`'. Note that, because '=' and '`!=`' are treated specially by the shell, you shouldn't quote the patterns: you might think that unless you do, they'll be turned into file names, but they won't. So

```
if [[ biryani = b* ]]; then
  print Word begins with a b.
fi
```

works. If you'd written '`b*`', including the quotes, it wouldn't have been treated as a pattern; it would have tested for a string which was exactly the two letters '`b*`' and nothing else. Pattern matching like this can be very powerful. If you've done any Bourne shell programming, you may remember the only way to use patterns there was via the '`case`' construction: that's still in zsh (see below), and uses the same sort of patterns, but the test form shown above is often more useful.

Then there are other three-argument tests which do numeric comparison. Rather oddly, these use letters rather than mathematical symbols: '`-eq`', '`-lt`' and '`-le`' compare if two numbers are equal, less than, or less than or equal, to one another. You can guess what '`-gt`' and '`-ge`' do. Note this is the other way round to Perl, which much more logically uses '==' to test for equality of numbers (not '=', since that's always an assignment operator in Perl) and '`eq`' (minus the minus) to test for equality of strings. Unfortunately we're now stuck with it this way round. If you are only comparing numbers, it's better to use the '`(( ... ))`' expression, because that has a proper understanding of arithmetic. However,

```
if [[ $number -gt 3 ]]; then
  print Wow, that\'s big
fi
```

and

```
if (( $number > 3 )); then
  print Wow, that\'s STILL big
fi
```

are essentially equivalent. In the second case, the status is zero (true) if the number in the expression was non-zero (sorry if I'm confusing you again) and vice versa. This means that

```
if (( 3 )); then
  print It seems that 3 is non-zero, Watson.
fi
```

is a perfectly valid test. As in C, the test operators in arithmetic return 1 for true and 0 for false, i.e. '`$number > 3`' is 1 if `$number` is greater than 3 and 0 otherwise; the inversion to shell logic, zero for true, only occurs at the final step when the expression has been completely evaluated and the '`(( ... ))`' command returns. At least with '`[[ ... ]]`' you don't need to worry about the extra negation; you can simply think in logical terms (although that's hard enough for a lot of people).

Finally, there are a few other odd comparisons in the three-argument form:

```
if [[ file1 -nt file2 ]]; then
  print file1 is newer than file2
fi
```

does the test implied by the example; there is also '`-ot`' to test for an older file, and there is also the little-used '`-ef`' which tests for an 'equivalent file', meaning that they refer to the same file — in other words, are linked; this can be a hard or a symbolic link, and in the second case it doesn't matter which of the two is the symbolic link. (If you were paying attention above, you'll know it can't possibly matter in the first case.)

In addition to these tests, which are pretty recognisable from most programming languages — although you'll just have to remember that the '=' family compares strings and not numbers — there are another set which are largely peculiar to UNIXy scripting languages. These are all in the form of a hyphen followed by a letter as the test, which always takes a single argument. I showed one: '-z $var' tests whether '`$var`' has zero length. It's opposite is '-n $var' which tests for non-zero length. Perhaps this is as good a time as any to point out that the arguments to these commands can be any single word expression, not just variables or filenames. You are quite at liberty to test

```
if [[ -z "$var is sqrt(`print bibble`)" ]]; then
  print Flying pig detected.
fi
```

if you like. In fact, the tests are so eager to make sure that they only have a one word argument that they will treat things like arrays, which usually return a whole set of words, as if they were in double quotes, joining the bits with spaces:

```
array=(two words)
if [[ $array = 'two words' ]]; then
  print "The array \$array is OK.  O, joy."
fi
```

Apart from '`-z`' and '`-n`', most of the two-argument tests are to do with files: '`-e`' tests that the file named next exists, whatever type of file it is (it might be a directory or something weirder); '`-f`' tests if it exists and is a regular file (so it isn't a directory or anything weird this time); '`-x`' tests whether you can execute it. There are all sorts of others which are listed in the manual page for various properties of files. Then there are a couple of others: '`-o <option>`' you've met and tests whether the option is set, and '`-t <fd>`' tests whether the file descriptor is attached to a terminal. A file descriptor is a number which for the shell must be between 0 and 9 inclusive (others may exist, but you can't access them directly); 0 is the standard input, 1 the standard output, and 2 the channel on which errors are usually printed. Hence '`[[ -t 0 ]]`' tests whether the input is coming from a terminal.

There are only four other things making up tests. '`&&`' and '`||`' mean logical 'and' and 'or', '`!`' negates the effect of a test, and parentheses '`( ... )`' can be used to surround a set of

tests which are to be treated as one. These are all essentially the same as in C. So

```
if [[ 3 -gt 2 && ( me > you || ! -z bah ) ]]; then
  print will I, won\'t I...
fi
```

will, because 3 is numerically greater than 2; the expression in parentheses is evaluated and though 'me' actually comes before 'you' in the alphabet, so the first test fails, '`-z bah`' is false because you gave it a non-empty string, and hence '`! -z bah`' is true. So both sides of the '`&&`' are true and the test succeeds.

3.2.13 Handling options to functions and scripts

It's often convenient to have your own functions and scripts process single-letter options in the way a lot of builtin commands (as well as a great many other UNIX-style commands) do. The shell provides a builtin for this called '`getopts`'. This should always be called in some kind of loop, usually a '`while`' loop. It's easiest to explain by example.

```
testopts() {
  # $opt will hold the current option
  local opt
  while getopts ab: opt; do
    # loop continues till options finished
    # see which pattern $opt matches...
    case $opt in
      (a)
         print Option a set
         ;;
      (b)
         print Option b set to $OPTARG
         ;;
# matches a question mark
# (and nothing else, see text)
      (\?)
         print Bad option, aborting.
         return 1
         ;;
    esac
  done
  (( OPTIND > 1 )) && shift $(( OPTIND - 1 ))
  print Remaining arguments are: $*
}
```

There's quite a lot here if you're new to shell programming. You might want to read the stuff on structures like `while` and `case` below and then come back and look at this. First let's see what it does.

```
% testopts -b foo -a -- args
Option b set to foo
Option a set
Remaining arguments are: args
```

Here's what's happening. '`getopts ab: opt`' is the argument to the '`while`'. That means that the `getopts` gets run; if it succeeds (returns status zero), then the loop after the '`do`' is run. When that's finished, the `getopts` command is run again, and so on until it fails (returns a non-zero status). It will do that when there are no more options left on the command line. So the loop processes the options one by one. Each time through, the number of the next argument to look at is left in the parameter `$OPTIND`, so this gradually increases; that's how `getopts` knows how far it has got.

The first argument to the `getopts` is '`ab:`'. That means '`a`' is an option which doesn't take an argument, while '`b`' is an argument which takes a single argument, signified by the colon after it. You can have any number of single-letter (or even digit) arguments, which are case-sensitive; for example '`ab:c:ABC:`' are six different options, three with arguments. If the option found has an argument, that is stored in the parameter `$OPTARG`; `getopts` then increments `$OPTIND` by however much is necessary, which may be 2 or just 1 since '`-b foo`' and '`-bfoo`' are both valid ways of giving the argument.

If an option is found, we use the '`case`' mechanism to find out what it was. The idea of this is simple, even if the syntax has the look of an 18th-century French chateau: the argument '`$opt`' is tested against all of the patterns in the '`pattern`)' lines until one matches, and the commands are executed until the next '`;;`'. It's the shell's equivalent of C's '`switch`'. In this example, we just print out what the `getopts` brought in. Note the last line, which is called if `$opt` is a question mark — it's quoted because '`?`' on its own can stand for any single character. This is how `getopts` signals an unknown option. If you try it, you'll see that `getopts` prints its own error message, so ours was unnecessary: you can turn the former off by putting a colon right at the start of the list of options, making it '`:ab:`' here.

Actually, having this last pattern as an *un*quoted '`?`' isn't such a bad idea. Suppose you add a letter to the list that `getopts` should handle and forget to add a corresponding item in the `case` list for it. If the last item matches any character, you will get the behaviour for an unhandled option, which is probably the best thing to do. Otherwise nothing in the `case` list will match, the shell will sail blithely on to the next call to `getopts`, and when you try to use the function with the new option you will be left wondering quite what happened to it.

The last piece of the `getopts` jigsaw is the next line, which tests if `$OPTIND` is larger than 1, i.e. an option was found and `$OPTIND` was advanced — it is automatically set to 1 at the start of every function or script. If it was, the '`shift`' builtin with a numeric argument, but no array name, moves the positional parameters, i.e. the function's arguments, to shift away the options that have been processed. The `print` in the next line shows you that only the remaining non-option arguments are left. You don't need to do that — you can just start using the remaining arguments from `$argv[$OPTIND]` on — but it's a pretty good way of doing it.

In the call, I showed a line with '`--`' in it: that's the standard way of telling `getopts` that the options are finished; even if later words start with a `-`, they are not treated as options. However, `getopts` stops anyway when it reaches a word not beginning with '`-`', so that wasn't necessary here. But it works anyway.

You can do all of what `getopts` does without *that* much difficulty with a few extra lines of shell programming, of course. The best argument for using `getopts` is probably that it allows you to group single-letter options, e.g. '`-abc`' is equivalent to '`-a -b -c`' if none of them was defined to have an argument. In this case, `getopts` has to remember the position *inside* the word on the command line for you to call it next, since the '`a`' '`b`' and '`c`' still appear on different calls. Rather unsatisfactorily, this is hidden inside the shell (as it is in other shells — we haven't fixed *all* of everybody else's bad design decisions); you can't get at it or reset it without altering `$OPTIND`. But if you read the small print at the top of the guide,

you'll find I carefully avoided saying everything was satisfactory.

While we're at it, why do blocks starting with 'if' and 'then' end with 'fi', and blocks starting with 'case' end with 'esac', while those starting with 'while' and 'do' end with 'done', not 'elihw' (perfectly pronounceable in Welsh, after all) or 'od'? Don't ask me.

3.2.14 Random file control things

We're now down into the odds and ends. If you know UNIX at all, you will already be familiar with the umask command and its effect on file creation, but as it is a builtin I will describe it here. Create a file and look at it:

```
% touch tmpfile
% ls -l tmpfile
-rw-r--r--    1 pws   pws    0 Jul 19 21:19 tmpfile
```

(I've shortened the output line for the TeX version of this document.) You'll see that the permissions are read for everyone, write-only for the owner. How did the command (touch, not a builtin, creates an empty file if there was none, or simply updates the modification time of an existing file) know what permissions to set?

```
% umask
022
% umask 077
% rm tmpfile; touch tmpfile
% ls -l tmpfile
-rw-------    1 pws   pws    0 Jul 19 21:22 tmpfile
```

umask was how it knew. It gives an octal number corresponding to the permissions which should *not* be given to a newly created file (only newly created files are affected; operations on existing files don't involve umask). Each digit is made up of a 4 for read, 2 for write, 1 for executed, in the same order that ls shows for permissions: user, then group, then everyone else. (On this Linux/GNU-based system, like many others, users have their own groups, so both are called 'pws'.) So my original '022' specified that everything should be allowed except writing for group and other, while '077' disallowed any operation by group and other. These are the two most common settings, although here '002' or '007' would be useful because of the way groups are specific to users, making it easier to grant permission to specific other users to write in my directories. (Except there aren't any other users.)

You can also use chmod-like permission changing expressions in umask. So

```
% umask go+rx
```

would restore group and other permissions for reading and executing, hence returning the mask to 022. Note that because it is *adding* permissions, just like chmod does, it is *removing* numbers from the umask.

You might have wondered about execute permissions, since 'touch' didn't give any, even where it was allowed by umask. That's because only operations which create executable programmes, such as running a compiler and linker, set that bit; the normal way of opening a new file — internally, the UNIX open function, with the O_CREAT flag set — doesn't touch

it. For the same reason, if you create shell scripts which you want to be able to execute by typing the name, you have to make them executable yourself:

```
% chmod +x myscript
```

and, indeed, you can think of `chmod` as `umask`'s companion for files which already exist. It doesn't need to be a builtin, because the files you are operating on are external to `zsh`; `umask`, on the other hand, operates when you create a file from within `zsh` or any child process, so needs to be a builtin. The fact that it's inherited means you can set `umask` before you start an editor, and files created by that editor will reflect the permissions.

Note that the value set by `umask` is also inherited and used by `chmod`. In the example of `chmod` I gave, I didn't see *which* type of execute permission to add; `chmod` looks at my `umask` and decides based on that — in other words, with 022, everybody would be allowed to execute `myscript`, while with 077, only I would, because of the 1's in the number: (0+0+0)+(4+2+1)+(4+2+1). Of course, you can be explicit with chmod and say '`chmod u+x myscript`' and so on.

Something else that may or may not be obvious: if you run a script by passing it as an argument to the shell,

```
% zsh myscript
```

what matters is *read* permission. That's what the shell's doing to the script to find the commands, after all. Execute permission applies when the system (or, in some cases, including zsh, the parent shell where you typed '`myscript`') has to decide whether to find a shell to run the script by itself.

3.2.15 Don't watch this space, watch some other

Finally for builtins, some things which really belong elsewhere. There are three commands you use to control the shell's editor. These will be described in chapter 4, where I talk all about the editor.

The `bindkey` command allows you to attach a set of keystrokes to a command. It understands an abbreviated notation for the keystrokes.

```
% bindkey '^Xc' copy-prev-word
```

This binds the keystrokes consisting of `Ctrl` held down with `x`, then `c`, to the command which copies the previous word on the line to the current position. The commands are listed in the `zshzle` manual page. `bindkey` can also do things with keymaps, which are a complete set of mappings between keys and commands like the one I showed.

The `vared` command is an extremely useful builtin for editing a shell variable. Usually much the easiest way to change `$path` (or `$PS1`, or whatever) is to run '`vared path`': note the lack of a '`$`', since otherwise you would be editing whatever `$path` was expanded to. This is because very often you want to leave most of what's there and just change the odd character or word. Otherwise, you would end up doing this with ordinary parameter substitutions, which are a lot more complicated and error prone. Editing a parameter is exactly like editing a command line, except without the prompt at the start.

Finally, there is the `zle` command. This is the most mysterious, as it offers a fairly low-level interface to the line editor; you use it to define your own editing commands. So I'll leave this alone for now.

3.2.16 And also

There is one more standard builtin that I haven't covered: `zmodload`, which allows you to manipulate add-on packages for zsh. Many extras are supplied with the shell which aren't normally loaded to keep down the use of memory and to avoid having too many rarely used builtins, etc., getting in the way. In the last chapter I will talk about some of these. To be more honest, a lot of the stuff in between actually uses these addons, generically referred to as modules — the line editor, zle, is itself a separate module, though heavily dependent on the main shell — and you've probably forgotten I mentioned above using '`zmodload zsh/mathfunc`' to load mathematical functions.

3.3 Functions

Now it's time to look at functions in more detail. The various issues to be discussed are: loading functions, handling parameters, compiling functions, and repairing bike tyres when the rubber solution won't stick to the surface. Unfortunately I've already used so much space that I'll have to skip the last issue, however topical it might be for me at the moment.

3.3.1 Loading functions

Well, you know what happens now. You can define functions on the command line:

```
fn() {
  print I am a function
}
```

which you call under the name '`fn`'. As you type, the shell knows that it's in the middle of a function definition, and prompts you until you get to the closing brace.

Alternatively, and much more normally, you put a file called `fn` somewhere in a directory listed in the `$fpath` array. At this point, you need to be familiar with the `KSH_AUTOLOAD` option described in the last chapter. From now on, I'm just going to assume your autoloadable function files contain just the body of the function, i.e. `KSH_AUTOLOAD` is not set. Then the file `fn` would contain:

```
print I am a function
```

and nothing else.

Recent versions of zsh, since `3.1.6`, set up `$fpath` for you. It contains two parts, although the second may have multiple directories. The first is, typically, `/usr/local/share/zsh/site-functions`, although the prefix may vary. This is empty unless your system administrator has put something in it, which is what it's there for.

The remaining part may be either a single directory such as `/usr/local/share/zsh/3.1.9/functions`, or a whole set of directories starting with that path. It simply depends whether the person installing zsh wanted to keep all the functions in the same directory, or have them sorted according to what they do. These directories are full of functions. However, none of the functions is autoloaded automatically, so unless you specifically put '`autoload ...`' in a startup file, the shell won't actually take any notice of them. As you'll see, part of the path is the shell version. This makes it very easy to keep multiple versions of zsh with functions which use features that may be different between the two versions. By the way, if these directories don't exist, you should check `$fpath` to see if they are in some other location, and if you can't find any correspondence between what's in `$fpath` and what's on the disk even when you start the shell with `zsh -f` to suppress loading of startup files, complain to the system administrator: he or she has either not installed them properly, or has made `/etc/zshenv` stomp on `$fpath`, both of which are thoroughly evil things to do. (`/etc/zshrc`, `/etc/zprofile` and `/etc/zlogin` shouldn't stomp on `$fpath` either, of course. In fact, they shouldn't do very much; that's up to the user.)

One point about `autoload` is the '`-U`' option. This turns off the use of any aliases you have defined when the function is actually loaded — the flag is remembered with the name of the function for future reference, rather than being interpreted immediately by the `autoload` command. Since aliases can pretty much redefine any command into any other, and are usually interpreted while a function is being defined or loaded, you can see that without this flag there is fair scope for complete havoc.

```
alias ls='echo Your ls command has been requisitioned.'
lsroot() {
  ls -l /
}
lsroot
```

That's not what the function writer intended. (Yes, I know it actually *is*, because I wrote it to show the problem, but that's not what I *meant*.) So `-U` is recommended for all standard functions, where you have no easy way of telling quite what could be run inside.

Recently, the functions for the new completion system (described in chapter 6) have been changing the fastest. They either begin with `comp` or an underscore, '`_`'. If the `functions` directory is subdivided, most of the subdirectories refer to this. There are various other classes of functions distributed with the shell:

- Functions beginning `zf` are associated with zftp, a builtin system for FTP transfers. Traditional FTP clients, ones which don't use a graphical interface, tend to be based around a set of commands on a command line — exactly what zsh is good at. This also makes it very easy to write macros for FTP jobs — they're just shell functions. This is described in the final chapter along with other modules. It's based around a single builtin, `zftp`, which is loaded from the module `zsh/zftp`.

- Functions beginning `prompt`, which may be in the `Prompts` subdirectory, are part of a 'prompt themes' system which makes it easy for you to switch between preexisting prompts. You load it with '`autoload -U promptinit; promptinit`'. Then '`prompt -h`' will tell you what to do next. If you have new completion loaded (with '`autoload -U compinit; compinit`', what else) the arguments to '`prompt`' can be listed with `^D` and completed with a TAB; they are various sorts of prompt which you may or may not like.

- Functions with long names and hyphens, like `predict-on` and `incremental-complete-word`. These are editor functions; you use them with

  ```
  zle -N predict-on
  bindkey <keystroke> predict-on
  ```

  Here, the `predict-on` function automatically looks back in the history list for matching lines as you type. You should also bind `predict-off`, which is loaded when `predict-on` is first called. `incremental-complete-word` is a fairly simple attempt at showing possible completions for the current word as you type; it could do with improving.

- Everything else; these may be in the `Misc` subdirectory. These are a very mixed bag which you should read to see if you like any. One of the most useful is `zed`, which allows you to edit a small file (it's really just a simple front-end to `vared`). The `run-help` file shows you the sort of thing you might want to define for use with the `\eh` (`run-help`) keystroke. `is-at-least` is a function for finding out if the version of the shell running is recent enough, assuming you know what version you need for a given feature. Several of the other functions refer to the old completion system — which you won't need, since you will be reading chapter 6 and using the new completion system, of course.

If you have your own functions — and if you use zsh a lot, you almost certainly will eventually — it's a good idea to add your own personal directory to the front of `$fpath`, so that everything there takes precedence over the standard functions. That allows you to override a completion function very easily, just by copying it and editing it. I tend to do something like this in my `.zshenv`:

```
[[ $fpath = *pws* ]] || fpath=(~pws/bin/fns $fpath)
```

to protect against the possibility that the directory I want to add is already there, in case I source that startup file again, and there are other similar ways. (You may well find your own account isn't called `pws`, however.)

Chances are you will always want your own functions to be autoloaded. There is an easy way of doing this: put it just after the line I showed above:

```
autoload ${fpath[1]}/*(:t)
```

The `${fpath[1]}/*` expands to all the files in the directory at the head of the `$fpath` array. The `(:t)` is a 'glob modifier': applied to a filename generation pattern, it takes the tail (basename) of all the files in the list. These are exactly the names of the functions you want to autoload. It's up to you whether you want the `-U` argument here.

3.3.2 Function parameters

I covered local parameters in some detail when I talked about `typeset`, so I won't talk about that here. I didn't mention the other parameters which are effectively local to a function, the ones that pass down the arguments to the function, so here is more detail. They work pretty much identically in scripts.

There are basically two forms. There is the form inherited from Bourne shell via Korn shell, with the typically uninformative names: `$#`, `$*`, `$@` and the numerical parameters `$1` etc. —

as high a number as the shell will parse is allowed, not just single digits. Then there is the form inherited from the C shell: `$ARGC` and `$argv`. I'll mainly use the Bourne shell versions, which are far more commonly used, and come back to some oddities of the C shell variety at the end.

`$#` tells you how many arguments were passed to the function, while `$*` gives those arguments as an array. This was the only array available in the Bourne shell, otherwise there would probably have been a more obvious way of doing it. To get the size and the number of elements of the array you don't use `${#*}` and `${*[1]}` etc. (well, you usually don't — zsh is typically permissive here), you use `$1`, `$2`. Despite the syntax, these are rather like ordinary array elements; if you refer to one off the end, you will get an empty string, but no error, unless you have the option `NO_UNSET` set. It is this not-quite array which gets shifted if you use the `shift` builtin without an argument: the old `$1` disappears, the old `$2` becomes `$1`, and so on, while `$#` is reduced by one. If there were no arguments (`$#` was zero), nothing happens.

The form `$@` is very similar to `$*`, and you can use it in place of that in most contexts. There is one place where they differ. Let's define a function which prints the number of its arguments, then the arguments.

```
args() {
  print $# $*
}
```

Now some arguments. We'll do this for the current shell — it's a slightly odd idea, that you can set the arguments for what's already running, or that an interactive shell has arguments at all, but nonetheless it's true:

```
set arguments to the shell
print $*
```

sets `$*` and hence prints the message '`arguments to the shell`'. We now pass *these* arguments on to the function in two different ways:

```
args $*
args $@
```

This outputs

```
4 arguments to the shell
4 arguments to the shell
```

– no surprises so far. Remember, too, that zsh doesn't split words on spaces unless you ask it too. So:

```
% set 'one word'
% args $*
1 one word
% args $@
1 one word
```

Now here's the difference:

```
% set two words
% args "$*"
1 two words
% args "$@"
2 two words
```

In quotes, `"$*"` behaves as a normal array, joining the words with spaces. However, `"$@"` doesn't — it still behaves as if it was unquoted. You can't see from the arguments themselves in this case, but you can from the digit giving the number of arguments the function has.

This probably seems pretty silly. Why quote something to have it behave like an unquoted array? The original answer lies back in Bourne shell syntax, and relates to the vexed question of word splitting. Suppose we turn on Bourne shell behaviour, and try the example of a word with spaces again:

```
% setopt shwordsplit
% set 'one word'
% args $*
2 one word
% args $@
2 one word
% args "$*"
1 one word
% args "$@"
1 one word
```

Aha! *This* time `"$@"` kept the single word with the space intact. In other words, `"$@"` was a slightly primitive mechanism for suppressing splitting of words, while allowing the splitting of arrays into elements. In zsh, you would probably quite often use `$*`, not `"$@"`, safe in the knowledge that nothing was split until you asked it to be; and if you wanted it split, you would use the special form of substitution `${=*}` which does that:

```
% unsetopt shwordsplit
% args $*
1 one word
% args ${=*}
2 one word
```

(I can't tell you why the '=' was chosen for this purpose, except that it consists of two split lines, or in an assignment it splits two things, or something.) This works with any parameter, whether scalar or array, quoted or unquoted.

However, that's actually not quite the whole story. There are times when the shell removes arguments, because there's nothing there:

```
% set hello '' there
% args $*
2 hello there
```

The second element of the array was empty, as if you'd typed

```
2=
```

— yes, you can assign to the individual positional parameters directly, instead of using `set`. When the array was expanded on the command line, the empty element was simply missed out altogether. The same happens with all empty variables, including scalars:

```
% empty=
% args $empty
0
```

But there are times when you don't want that, any more than you want word splitting — you want *all* arguments passed just as you gave them. This is another side effect of the `"$@"` form.

```
% args "$@"
3 hello there
```

Here, the empty element was passed in as well. That's why you often find `"$@"` being used in zsh when wordsplitting is already turned off.

Another note: why does the following not work like the example with `$*`?

```
% args hello '' there
3 hello there
```

The quotes were kept here. Why? The reason is that the shell doesn't elide an argument if there were quotes, even if the result was empty: instead, it provides an empty string. So this empty string was passed as the second argument. Look back at:

```
set hello '' there
```

Although you probably didn't think about it at the time, the same thing was happening here. Only with the `''` did we get an empty string assigned to `$2`; later, this was missed out when passing `$*` to the function. The same difference occurs with scalars:

```
% args $empty
0
% args "$empty"
1
```

The `$empty` expanded to an empty string in each case. In the first case it was unquoted and was removed; this is like passing an empty part of `$*` to a command. In the second case, the quotes stopped that from being removed completely; this is similar to setting part of `$*` to an empty string using `″`.

That's all thoroughly confusing the first time round. Here's a table to try and make it a bit clearer.

```
                        |   Number of arguments
                        |     if $* contains...
                        |  (two words)
Expression   Word       |       'one word'
on line   splitting?    |             empty string
```

```
-------------------------------------------------
$*              n       |       2       1       0
$@              n       |       2       1       0
"$*"            n       |       1       1       1
"$@"            n       |       2       1       1
                        |
$*              y       |       2       2       0
$@              y       |       2       2       0
"$*"            y       |       1       1       1
"$@"            y       |       2       1       1
                        |
${=*}           n       |       2       2       0
${=@}           n       |       2       2       0
"${=*}"         n       |       2       2       1
"${=@}"         n       |       2       2       1
```

On the left is shown the expression to be passed to the function, and in the three right hand columns the number of arguments the function will get if the positional parameters are set to an array of two words, a single word with a space in the middle, or a single word which is an empty string (the effect of '`set - ''`') respectively. The second column shows whether word splitting is in effect, i.e. whether the `SH_WORD_SPLIT` option is set. The first four lines show the normal zsh behaviour; the second four show the normal sh/ksh behaviour, with word splitting turned on — only the case where a word has a space in it changes, and then only when no quotes are supplied. The final four show what happens when you use the '`${=..}`' method to turn on word splitting, for convenience: that's particularly simple, since it always forces words to be split, even inside quotation marks.

I would recommend that anyone not wedded to the Bourne shell behaviour use the top set as standard: in particular, '`$*`' for normal array behaviour with removal of empty items, '`"$@"`' for normal array behaviour with empty items left as empty items, and '`"$*"`' for turning arrays into single strings. If you need word-splitting, you should use '`${=*}`' or '`"${=@}"`' for splitting with/without removal of empty items (obviously there's no counterpart to the quoted-array behaviour here). Then keep `SH_WORD_SPLIT` turned off. If you are wedded to the Bourne shell behaviour, you're on your own.

It's a bug

There's a bug in handling of the the form `${1+"$@"}`. This looks rather an arcane and unlikely combination, but actually it is commonly used to get around a bug in some versions of the Bourne shell (which is not in zsh): that `"$@"` generates a single empty argument if there are no arguments. The form shown tests whether there is a first argument, and if so substitutes `"$@"`, else it doesn't substitute anything, avoiding the bug.

Unfortunately, in zsh, when `shwordsplit` is set — which is the time you usually run across attempts like this to standardise the way different shells work — this will actually cause too much word-splitting. The way the shell is written at the moment, the embedded `"$@"` will force extra splitting on spaces inside the arguments. So if the first argument is '`one word`', and `shwordsplit` is set, `${1+"$@"}` produces *two* words '`one`' and '`word`'.

Oliver Kiddle spotted a way of getting round this which has been adapted for use in the GNU autoconf package: in your initialisation code, have

```
[ x$ZSH_VERSION != x ] && alias -g '${1+"$@"}'='"$@"'
```

This uses a global alias to turn `${1+"$@"}` wherever it occurs as a single word into `"$@"` which doesn't have the problem. Aliasing occurs so early in processing that the fact that most of the characters have a special meaning to the shell is irrelevant; the shell behaves as if it read in `"$@"`. The only catch is that for this to work the script or function must use *exactly* the character string `${1+"$@"}`, with no leading or trailing word characters (whitespace, obviously, or characters which terminate parsing such as '`;`' are all right). Some day, we may fix the underlying bug, but it's not very easy with the way the parameter substitution code is written at the moment.

Parameters inherited from csh

The final matter is the C-shell syntax. There are two extra variables but, luckily, there is not much extra in the way of complexity. `$ARGC` is essentially identical to `$#`, and `$argv` corresponds to `$*`, but is a real array this time, so instead of `$1` you have `${argv[1]}` and so on. They use the convention that scalars used by the shell are all uppercase, while arrays are all lowercase. This feature is probably the only reason anyone would need these variants. For example, `${argv[2,-1]}` means all arguments from the second to the last, inclusive: negative indices count from the end, and a comma indicates a slice of an array, so that `${argv[1,-1]}` is always the same as the full array. Otherwise, my advice would be to stick with the Bourne shell variants, however cryptic they may look at first sight, for the usual reason that zsh isn't really like the C shell and if you pretend it is, you will come a cropper sooner or later.

It looks like you're missing `"$@"`, but actually you can do that with `"${argv[@]}"`. This, like negative indices and slices, works with all arrays.

There's one slight oddity with `$ARGC` and `$argv`, which isn't really a deliberate feature of the shell at all, but just in case you run into it: although the values in them are of course local to functions, the variables `$ARGC` and `$argv` *themselves* are actually treated like global variables. That means if you apply a `typeset -g` command to them, it will affect the behaviour of `$ARGC` and `$argv` in all functions, even though they have different values. It's probably not a good idea to rely on this behaviour.

nusubsect(Arguments to all commands work the same)

I've been a little tricky here, because I've been talking about two levels of functions at once: `$*` and friends as set in the current function, or even at the top level, as well as how they are passed down to commands such as my `args` function. Of course, in the second case the same behaviour applies to all commands, not just functions. What I mean is, in

```
fn() {
  cat $*
  cat "$*"
}
```

the '`cat`' command will see the differences in behaviour between the two calls just as `args` would. That should be obvious.

It's not a bug

Let me finally mention again a feature I noted in passing:

```
1='first argument'
```

sets the first command argument for the current shell or function, independently of any others. People sometimes complain that

```
1000000='millionth argument'
```

suddenly makes the shell use a lot more memory. That's not a bug at all: you've asked the shell to set the millionth element of an array, but not any others, so the shell creates an array a million elements long with the first 999,999 empty, except for any arguments which were already set. It's not surprising this takes up a lot of memory.

3.3.3 Compiling functions

Since version 3.1.7, it has been possible to compile functions to their internal format. It doesn't make the functions run any faster, it just reduces their loading time; the shell just has to bring the function into memory, then it 'runs it as it does any other function. On many modern computers, therefore, you don't gain a great deal from this. I have to admit I don't use it, but there are other definite advantages.

Note that when I say 'compiled' I don't mean the way a C compiler, say, would take a file and turn it into the executable code which the processor understands; here, it's simply the format that the shell happens to use internally — it's useless without a suitable version of zsh to run it. Also, it's no use thinking you can hide your code from prying eyes this way, like you can to some extent with an ordinary compiler (disassembling anything non-trivial from scratch being a time-consuming job): first of all, ordinary command lines appear inside the compiled files, except in slightly processed form, and secondly running '`functions`' on a compiled function which has been loaded will show you just as much as it would if the function had been loaded normally.

One other advantage is that you can create 'digest' files, which are sets of functions stored in a single file. If you often use a large fraction of those files, or they are small, or you like the function itself to appear when you run 'functions' rather than a message saying it hasn't been loaded, then this works well. In fact, you can compile all the functions in a single directory in one go. You might think this uses a lot of memory, but often zsh will simply 'memory map' the file, which means rather than reserving extra main memory for it and reading it in — the obvious way of reading files — it will tell the operating system to make the file available as if it were memory, and the system will bring it into memory piece by piece, 'paging' the file as it is needed. This is a very efficient way of doing it. Actually, zsh supports both this method and the obvious method of simply reading in the file (as long as your operating system does); this is described later on.

A little extra, in case you're interested: if you read in a file normally, the system will usually reserve space on a disk for it, the 'swap', and do paging from there. So in this case you still get the saving of main memory — this is standard in all modern operating systems. However, it's not *as* efficient: first of all, you had to read the file in in the first place. Secondly it eats up swap space, which is usually a fixed amount of disk, although if you've got enough main memory, the system probably won't bother allocating swap. Thirdly — this is probably the clincher for standard zsh functions on a large system — if the file is directly mapped read-only, as it is in this case, the system only needs one place in main memory, plus the single original file on disk, to keep the function, which is very much more efficient. With the other method, you would get multiple copies in both main memory and (where necessary) swap.

This is how the system treats directly executable programmes like the shell itself — the data is specific to each process, but the programme itself can be shared because it doesn't need to be altered when it's running.

Here's a simple example.

```
% echo 'echo hello, world' >hw
% zcompile hw
% ls
hw    hw.zwc
% rm hw
% fpath=(. $fpath)
% autoload hw
% hw
hello, world
```

We created a simple 'hello, world' function, and compiled it. This produces a file called '`hw.zwc`'. The extension stands for 'Z-shell Word Code', because it's based on the format of words (integers longer than a single byte) used internally by the shell. Then we made sure the current directory was in our `$fpath`, and autoloaded the function, which ran as expected. We deleted the original file for demonstration purposes, but as long as the '`.zwc`' file is newer, that will be used, so you don't need to remove the originals in normal use. In fact, you shouldn't, because you will lose any comments and formatting information in it; you can regenerate the function itself with the '`functions`' command (try it here), but the shell only remembers the information actually needed to run the commands. Note that the function was in the zsh autoload format, not the ksh one, in this case (but see below).

And there's more

Now some bells and whistles. Remember the `KSH_AUTOLOAD` thing? When you compile a function, you can specify which format — native zsh or ksh emulation — will be used for loading it next time, by using the option `-k` or `-z`, instead of the default, which is to examine the option (as would happen if you were autoloading directly from the file). Then you don't need to worry about that option. So, for example, you could compile all the standard zsh functions using '`zcompile -z`' and save people the trouble of making sure they are autoloaded correctly.

You can also specify that aliases shouldn't be expanded when the files are compiled by using `-U`: this has roughly the same effect as saying `autoload -U`, since when the shell comes to load a compiled file, it will never expand aliases, because the internal format assumes that all processing of that kind has already been done. The difference in this case is if you *don't* specify `-U`: then the aliases found when you compile the file, not when you load the function from it, will be used.

Now digest files. Here's one convenient way of doing it.

```
% ls ~/tmp/fns
hw1   hw2
% fpath=(~/tmp/fns $fpath)
% cd ~/tmp
% zcompile fns fns/*
% ls
fns   fns.zwc
```

We've made a directory to put functions in, `~/tmp/fns`, and stuck some random files in it. The `zcompile` command, this time, was given several arguments: a filename to use for the compiled functions, and then a list of functions to compile into it. The new file, `fns.zwc`, sits in the same directory where the directory `fns`, found in `$fpath`, is. The shell will actually search the digest file instead of the directory. More precisely, it will search both, and see which is the more recent, and use that as the function. So now

```
% autoload hw1
% hw1
echo hello, first world
```

You can test what's in the digest file with:

```
% zcompile -t fns
zwc file (read) for zsh-3.1.9-dev-3
fns/hw1
fns/hw2
```

Note that the names appear as you gave them on the command line, i.e. with `fns/` in front. Only the basenames are important for autoloading functions. The note '`(read)`' in the first line means that zsh has marked the functions to be read into the shell, rather than memory mapped as discussed above; this is easier for small functions, particularly if you are liable to remove or alter a file which is mapped, which will confuse the shell. It usually decides which method to use based on size; you can force memory mapping by giving the `-M` option. Memory mapping doesn't work on all systems (currently including Cygwin).

I showed this for compiling files, but you can actually tell the shell to output compiled functions — in other words, it will look along `$fpath` and compile the functions you specify. I find compiling files easier, when I do it at all, since then I can use patterns to find them as I did above. But if you want to do it the other way, you should note two other options: `-a` will compile files by looking along `$fpath`, while `-c` will output any functions already loaded by the shell (you can combine the two to use either). The former is recommended, because then you don't lose any information which was present in the autoload file, but not in the function stored in memory —- this is what would happen if the file defined some extra widgets (in the non-technical sense) which weren't part of the function called subsequently.

If you're perfectly happy with the shell *only* searching a digest file, and not comparing the datestamp with files in the directory, you can put that directly into your `$fpath`, i.e. `~/tmp/fns.zwc` in this case. Then you can get rid of the original directory, or archive it somewhere for reuse.

You can compile scripts, too. Since these are in the same format as a zsh autoload file, you don't need to do anything different from compiling a single function. You then run (say) `script.zwc` by typing '`zsh script`' — note that you should omit the `.zwc`, as zsh decides if there's a compiled version of a script by explicitly appending the suffix. What's more, you can run it using '`.`' or '`source`' in just the same way ('`. script`') — this means you can compile your startup files if you find they take too long to run through; the shell will spot a `~/.zshrc.zwc` as it would any other sourceable file. It doesn't make much sense to use the memory mapping method in this case, since once you've sourced the files you never want to run them again, so you might as well specify '`zcompile -R`' to use the reading (non-memory-mapping) method explicitly.

If you ever look inside a `.zwc` file, you will see that the information is actually included twice. That's because systems differ about the order in which numbers are stored: some have

the least significant byte first (notably Intel and some versions of Mips) and some the most significant (notably SPARC and Cambridge Consultants' XAP processor, which is notable here mainly because I spend my working hours programming for it — you can't run zsh on it). Since zsh uses integers a great deal in the compiled code, it saves them in both possible orders for ease of use. Why not just save it for the machine where you compiled it? Then you wouldn't be able to share the files across a heterogeneous network — or even worse, if you made a distribution of compiled files, they would work on some machines, and not on others. Think how Emacs users would complain if the `.elc` files that arrived weren't the right ones. (Worse, think how the vi users would laugh.) The shell never reads or maps in the version it doesn't use, however; only extra disk space is used.

A little -Xtra help

There are two final autoloading issues you might want to know about. In versions of zsh since 3.1.7, you will see that when you run `functions` on a function which is marked for autoload but hasn't yet been loaded, you get:

```
afunctionmarkedforautoloadwhichhasntbeenloaded () {
        # undefined
        builtin autoload -XU
}
```

The '`# undefined`' is just printed to alert you that this was a function marked as autoloadable by the `autoload` command: you can tell, because it's the only time `functions` will emit a comment (though there might be other '`#`' characters around). What's interesting is the `autoload` command with the `-X` option. That option means 'Mark me for autoloading and run me straight away'. You can actually put it in a function yourself, and it will have the same effect as running '`autoload`' on a not-yet-existent function. Obviously, the `autoload` command will disappear as soon as you do run it, to be replaced by the real contents. If you put this inside a file to be autoloaded, the shell will complain — the alternative is rather more unpalatable.

Note also the `-U` option was set in that example: that simply means that I used `autoload` with the `-U` option when I originally told the shell to autoload the function.

There's another option, `+X`, the complete opposite of `-X`. This one can *only* be used with autoload outside the function you're loading, just as `-X` was only meaningful inside. It means 'load the file immediately, but don't run it', so it's a more active (or, as they say nowadays, since they like unnecessarily long words, proactive) form of `autoload`. It's useful if you want to be able to run the `functions` command to see the function, but don't want to run the function itself.

Special functions

I'm in danger of simply quoting the manual, but there are various functions with a special meaning to the shell (apart from `TRAP...` functions, which I've already covered). That is, the functions themselves are perfectly normal, but the shell will run them automatically on certain occasions if they happen to exist, and silently skip them if they don't.

The two most frequently used are `chpwd` and `precmd`. The former is called whenever the directory changes, either via `cd`, or `pushd`, or an `AUTO_CD` — you could turn the first two

into functions, and avoid needing `chpwd` but not the last. Here's how to force an xterm, or a similar windowing terminal, to put the current directory into the title bar.

```
chpwd() {
  [[ -t 1 ]] || return
  case $TERM in
    (sun-cmd) print -Pn "\e]l%~\e\\"
      ;;
    (*xterm*|rxvt|(dt|k|E)term) print -Pn "\e]2;%~\a"
      ;;
  esac
}
```

The first line tests that standard output is really a terminal — you don't want to print the string in the middle of a script which is directing its output to a file. Then we look to see if we have a `sun-cmd` terminal, which has its own *sui generis* sequence for putting a string into the title bar, or something which recognises xterm escape sequences. In either case, the special sequences (a bit like termcap sequences as discussed for `echotc`) are interpreted by the terminal, and instead of being printed out cause it to put the string in the middle into the title bar. The string here is '`%~`': I added the `-P` option to `print` so it would expand prompt escapes. I could just have used `$PWD`, but this way has the useful effect of shortening your home directory, or any other named directory, into `~`-notation, which is a bit more readable. Of course, you can put other stuff there if you like, or, if you're really sophisticated, put in a parameter `$HEADER` and define that elsewhere.

If programmes other than the shell alter what appears in the xterm title bar, you might consider changing that `chwpd` function to `precmd`. The function `precmd` is called just before every prompt; in this case it will restore the title line after every command has run. Some people make the mistake of using it to set up a prompt, but there are enough ways of getting varying information into a fixed prompt string that you shouldn't do that unless you have *very* odd things in your prompt. It's a big nuisance having to redefine `precmd` to alter your prompt — especially if you don't know it's there, since then your prompt apparently magically returns to the same format when you change it. There are some good reasons for using `precmd`, too, but most of them are fairly specialised. For example, on one system I use it to check if there is new input from a programme which is sending data to the shell asynchronously, and if so printing it out onto the terminal. This is pretty much what happens with job control notification if you don't have the `NOTIFY` option set.

The name `precmd` is a bit of a misnomer: `preprompt` would have been better. It usurps the name more logically applied to the function actually called `preexec`, which is run after you finished editing a command line, but just before the line is executed. `preexec` has one additional feature: the line about to be executed is passed down as an argument. You can't alter what's going to be executed by editing the parameter, however: that has been suggested as an upgrade, but it would make it rather easy to get the shell into a state where you can't execute any commands because `preexec` always messes them up. It's better, where possible, to write function front-ends to specific commands you want to handle specially. For example, here's my `ls` function:

```
local ls
if [[ -n $LS_COLORS ]]; then
  ls=(ls --color=auto)
else
  ls=(ls -F)
fi
```

```
command $ls $*
```

This handles GNU and non-GNU versions of ls. If `$LS_COLORS` is set, it assumes we are using GNU ls, and hence colouring (or colorizing, in geekspeak) is available. Otherwise, it uses the standard option `-F` to show directories and links with a special symbol. Then it uses `command` to run the real `ls` — this is a key thing to remember any time you use a function front-end to a command. I could have done this another way: test in my initialisation files which version of `ls` I was using, then alias `ls` to one of the two forms. But I didn't.

Apart from the trap functions, there is one remaining special function. It is `periodic`, which is executed before a prompt, like `precmd`, but only every now and then, in fact every `$PERIOD` seconds; it's up to you to set `$PERIOD` when you defined `periodic`. If `$PERIOD` isn't set, or is zero, nothing happens. Don't get `$PERIOD` confused with `$SECONDS`, which just counts up from 0 when the shell starts.

3.4 Aliases

Aliases are much simpler than functions. In the C shell and its derivatives, there are no functions, so aliases take their place and can have arguments, which involve expressions rather like those which extract elements of previous history lines with '`!`'. Zsh's aliases, like ksh's, don't take arguments; you have to use functions for that. However, there are things aliases can do which functions can't, so sometimes you end up using both, for example

```
zfget() {
  # function to retrieve a file by FTP,
  # using globbing on the remote host
}
alias zfget='noglob zfget'
```

The function here does the hard work; this is a function from the zftp function suite, supplied with the shell, which retrieves a file or set of files from another machine. The function allows patterns, so you can retrieve an entire directory with '`zfget *`'. However, you need to avoid the '`*`' being expanded into the set of files in the current directory on the machine you're logged into; this is where the alias comes in, supplying the '`noglob`' in front of the function. There's no way of doing this with the function alone; by the time the function is called, the '`*`' would already have been expanded. Of course you could quote it, but that's what we're trying to avoid. This is a common reason for using the alias/function combination.

Remember to include the '=' in alias definition, necessary in zsh, unlike csh and friends. If you do:

```
alias zfget noglob zfget
```

they are treated as a list of aliases. Since none has the '=' and a definition, the shell thinks you want to list the definitions of the listed words; I get the output

```
zfget='noglob zfget'
zfget='noglob zfget'
```

since `zfget` was aliased as before, but `noglob` wasn't aliased and was skipped, although the failed alias lookup caused status 1 to be returned. Remember that the `alias` command

takes as many arguments as you like; any with '=' is a definition, any without is a request to print the current definition.

Aliases can in fact be allowed to expand to almost anything the shell understands, not just sets of words. That's because the text retrieved from the alias is put back into the input, and reread more or less as if you'd typed it. That means you can get away with strange combinations like

```
alias tripe="echo foo | sed 's/foo/bar/' |"
tripe cat
```

which is interpreted exactly the same way as

```
echo foo | sed 's/foo/bar/' | cat
```

where the word '`foo`' is sent to the stream editor, which alters it to '`bar`' ('`s/old/new/`' is `sed`'s syntax for a substitution), and passes it on to '`cat`', which simply dumps the output. It's useless, of course, but it does show what can lurk behind an apparently simple command if it happens to be an alias. It is usually not a good idea to do this, due to the potential confusion.

As the manual entry explains, you can prevent an alias from being expanded by quoting it. This isn't like quoting any other expansion, though; there's no particular important character which has to be interpreted literally to stop the expansion. The point is that because aliases are expanded early on in processing of the command line, looking up an alias is done on a string without quotes removed. So if you have an alias '`drivel`', none of the strings '`\drivel`', '`'d'rivel`', or '`drivel""`' will be expanded as the alias: they all would have the same effect as proper commands, after the quotes are removed, but as aliases they appear different. The manual entry also notes that you can actually make aliases for any of these special forms, e.g. '`alias '\drivel'=...`' (note the quotes, since you need the backslash to be passed down to the alias command). You would need a pretty good reason to do so.

Although my '`tripe`' example was silly, you know from the existence of 'precommand modifiers' that it's sometimes useful to have a special command which precedes a command line, like `noglob` or the non-shell command `nice`. Since they have commands following, you would probably expect aliases to be expanded there, too. But this doesn't work:

```
% alias foo='echo an alias for foo'
% noglob foo
zsh: command not found: foo
```

because the `foo` wasn't in command position. The way round this is to use a special feature: aliases whose definitions end in a space force the next word along to be looked up as a possible alias, too:

```
% alias noglob='noglob '
% noglob foo
an alias for foo
```

which is useful for any command which can take a command line after it. This also shows another feature of aliases: unlike functions, they remember that you have already called an alias of a particular name, and don't look it up again. So the '`noglob`' which comes from expanding the alias is not treated as an alias, but as the ordinary precommand modifier.

You may be a little mystified about this difference. A simple answer is that it's useful that way. It's sometimes useful for functions to call themselves; for example if you are handling a directory hierarchy in one go you might get a function to examine a directory, do something for every ordinary file, and for every directory file call itself with the new directory name tacked on. Aliases are too simple for this to be a useful feature. Another answer is that it's particularly easy to mark aliases as being 'in use' while they are being expanded, because it happens while the strings inside them are being examined, before any commands are called, where things start to get complicated.

Lastly, there are 'global aliases'. If aliases can get you into a lot of trouble, global aliases can get you into a lot of a lot of trouble. They are defined with the option `-g` and are expanded not just in command position, but anywhere on the command line.

```
alias -g L='| less'
echo foo L
```

This turns into '`echo foo |less`'. It's a neat trick if you don't mind your command lines having only a minimal amount to do with what is actually executed.

I already pointed out that alias lookups are done so early that aliases are expanded when you define functions:

```
% alias hello='echo I have been expanded'
% fn() {
function>  hello
function> }
% which fn
fn () {
        echo I have been expanded
}
```

You can't stop this when typing in functions directly, except by quoting part of the name you type. When autoloading, the `-U` option is available, and recommended for use with any non-trivial function.

A brief word about that '`function>`' which appears to prompt you while you are editing a function; I mentioned this in the previous chapter but here I want to be clearer about what's going on. While you are being prompted like that, the shell is not actually executing the commands you are typing in. Only when it is satisfied that it has a complete set of commands will it go away and execute them (in this case, defining the function). That means that it won't always spot errors until right at the end. Luckily, zsh has multi-line editing, so if you got it wrong you should just be able to hit up-arrow and edit what you typed; hitting return will execute the whole thing in one go. If you have redefined `$PS2` (or `$PROMPT2`), or you have an old version of the shell, you may not see the full prompt, but you will usually see something ending in '>' which means the same.

3.5 Command summary

As a reminder, the shell looks up commands in this order:

- aliases, which will immediately be interpreted again as texts for commands, possible even other aliases; they can be deleted with '`unalias`',

- reserved words, those special to the shell which often need to be interpreted differently from ordinary commands due to the syntax, although they can be disabled if you really need to,
- functions; these can also be disabled, although it's usually easier to '`unfunction`' them,
- builtin commands, which can be disabled, or called as a builtin by putting '`builtin`' in front,
- external commands, which can be called as such, even if the name clashes with one of the above types, by putting '`command`' in front.

3.6 Expansions and quotes

As I keep advertising, there will be a whole chapter dedicated to the subject of shell expansions and what to do with them. However, it's a rather basic subject, which definitely comes under the heading of basic shell syntax, so I shall here list all the forms of expansion. As given in the manual, there are five stages.

3.6.1 History expansion

This is the earliest, and is only done on an interactive command line, and only if you have not set `NO_BANG_HIST`. It was described in the section '*The history mechanism; types of history*' in the previous chapter. It is almost independent of the shell's processing of the command line; it takes place as the command line is read in, not when the commands are interpreted. However, in zsh it is done late enough that the '`!`'s can be quoted by putting them in single quotes:

```
echo 'Hello!!'
```

doesn't insert the previous line at that point, but

```
echo "Hello!!"
```

does. You can always quote active '`!`'s with a backslash, so

```
echo "Hello\!\!"
```

works, with or without the double quotes. Amusingly, since single quotes aren't special in double quotes, if you set the `HIST_VERIFY` option, which puts the expanded history line back on the command line for possible further editing, and try the first two of the three possibilities above in order, then keep hitting return, you will find ever increasing command lines:

```
% echo 'Hello!!'
Hello!!
% echo "Hello!!"
% echo "Helloecho 'Hello!!'"
% echo "Helloecho 'Helloecho 'Hello!!''"
% echo "Helloecho 'Helloecho 'Helloecho 'Hello!!'''"
```

and if you understand why, you have a good grasp of how quotes work.

There's another way of quoting exclamation marks in a line: put a '`!"`' in it. It can appear anywhere (as long as it's not in single quotes) and will be removed from the line, but it has the effect of disabling any subsequent exclamation marks till the end of the line. This is the only time quote marks which are significant to the shell (i.e. are not themselves quoted) don't have to occur in a matching pair.

Note that as exclamation marks aren't active in any text read non-interactively — and this includes autoloaded functions and sourced files, such as startup files, read inside interactive shells — it is an error to quote any '`!`'s in double quotes in files. This will simply pass on the backslashes to the next level of parsing. Other forms of quoting are all right: '`\!`', because any character quoted with a backslash is treated as itself, and `'!'` because single quotes can quote anything anyway.

3.6.2 Alias expansion

As discussed above, alias expansion also goes on as the command line is read, so is to a certain extent similar to history expansion. However, while a history expansion may produce an alias for expansion, '`!`'s in the text resulting from alias expansions are normal characters, so it can be thought of as a later phase (and indeed it's implemented that way).

3.6.3 Process, parameter, command, arithmetic and brace expansion

There are a whole group of expansions which are done together, just by looking at the line constructed from the input after history and alias expansion and reading it from left to right, picking up any active expansions as the line is examined. Whenever a complete piece of expandable text is found, it is expanded; the text is not re-examined, except in the case of brace expansion, so none of these types of expansion is performed on any resulting text. Whether later forms of expansion — in other words, filename generation and filename expansion are performed — is another matter, depending largely on the `GLOB_SUBST` option as discussed in the previous chapter. Here's a brief summary of the different types.

Process substitution

There are three forms that result in a command line argument which refers to a file from or to which input or output is taken: '<(*process*)' runs the process which is expected to generate output which can be used as input by a command; '>(*process*)' runs the process which will take input to it; and '=(*process*)' acts like the first one, but it is guaranteed that the file is a plain file.

This probably sounds like gobbledygook. Here are some simple examples.

```
cat < <(echo This is output)
```

(There are people in the world with nothing better to do than compile lists of dummy uses of the '`cat`' command, as in that example, and pour scorn on them, but I'll just have to brave it out.) What happens is that the command '`echo This is output`' is run, with the obvious result. That output is *not* put straight into the command line, as it would be with command

substitution, to be described shortly. Instead, the command line is given a filename which, when read, gets that output. So it's more like:

```
echo This is output >tmpfile
cat < tmpfile
rm tmpfile
```

(note that the temporary file is cleaned up automatically), except that it's more compact. In this example I could have missed out the remaining '<', since `cat` does the right thing with a filename, but I put it there to emphasise the fact that if you want to redirect input from the process substitution you need an *extra* '<', over and above the one in the substitution syntax.

Here's an example for the corresponding output substitution:

```
echo This is output > \
>(sed 's/output/rubbish/' >outfile)
```

which is a perfectly foul example, but works essentially like:

```
echo This is output >tmpfile
sed 's/output/rubbish/' <tmpfile >outfile
```

There's an obvious relationship to pipes here, and in fact this example could be better written,

```
echo This is output | sed 's/output/rubbish/' >outfile
```

A good example of an occasion where the output process substitution can't be replaced by a pipe is when it's on the error output, and standard output is being piped:

```
./myscript 2> >(grep -v idiot >error.log) |
    process-output >output.log
```

a little abstract, but here the main point of the script 'myscript' is to produce some output which undergoes further processing on the right-hand side of the pipe. However, we want to process the error output here, by filtering out occurrences of lines which use the word 'idiot', before dumping those errors into a file `error.log`. So we get an effect similar to having two pipelines at once, one for output and one for error. Note again the *two* '>' signs present next to one another to get that effect.

Finally, the '=(*process*)' form. Why do we need this as well as the one with '<'? To understand that, you need to know a little of how zsh tries to implement the latter type efficiently. Most modern UNIX-like systems have 'named pipes', which are essentially files that behave like the '|' on the command line: one process writes to the file, another reads from it, and the effect is essentially that data goes straight through. If your system has them, you will usually find the following demonstration works:

```
% mknod tmpfile p
% echo This is output >tmpfile &
[2] 1507
% read line <tmpfile
%
```

```
[2]  + 1507 done       echo This is output >> tmpfile
% print -- $line
This is output
%
```

The syntax to create a named pipe is that rather strange '`mknod`' command, with '`p`' for pipe. We stick this in the background, because it won't do anything yet: you can't write to the pipe when there's no-one to read it (a fundamental rule of pipes which isn't *quite* as obvious as it may seem, since it *is* possible for data to lurk in the pipe, buffered, before the process reading from it extracts it), so we put that in the background to wait for action. This comes in the next line, where we read from the pipe: that allows the `echo` to complete and exit. Then we print out the line we've read.

The problem with pipes is that they are just temporary storage spaces for data on the way through. In particular, you can't go back to the beginning (in C-speak, 'you can't seek backwards on a pipe') and re-read what was there. Sometimes this doesn't matter, but some commands, such as editors, need that facility. As the '<' process substitution is implemented with named pipes (well, maybe), there is also the '=' form, which produces a real, live temporary file, probably in the '`/tmp`' directory, containing the output from the file, and then puts the name of that file on the command line. The manual notes, unusually helpfully, that this is useful with the '`diff`' command for comparing the output of two processes:

```
diff =(./myscript1) =(./myscript2)
```

where, presumably, the two scripts produce similar, but not identical, output which you want to compare.

I said 'well, maybe' in that paragraph because there's another way zsh can do '<' process substitutions. Many modern systems allow you to access a file with a name like '`/dev/fd/0`' which corresponds to file descriptor 0, in this case standard input: to anticipate the section on redirection, a 'file descriptor' is a number assigned to a particular input or output stream. This method allows you to access it as a file; and if this facility is available, zsh will use it to pass the name of the file in process substitution instead of using a named pipe, since in this case it doesn't have to create a temporary file; the system does everything. Now, if you are really on the ball, you will realise that this doesn't get around the problem of pipes — where is data on this file descriptor going to come from? The answer is that it will either have to come from a real temporary file — which is pointless, because that's what we wanted to avoid — or from a pipe opened from some process — which is equivalent to the named pipe method, except with just a file descriptor instead of a name. So even if zsh does it this way, you still need the '=' form for programmes which need to go backwards in what they're reading.

Parameter substitution

You've seen enough of this already. This comes from a '`$`' followed either by something in braces, or by alphanumeric characters forming the name of the parameter: '`$foo`' or '`${foo}`', where the second form protects the expansion from any other strings at the ends and also allows a veritable host of extra things to appear inside the braces to modify the substitution. More detail will be held over to till chapter 5; there's a lot of it.

Command substitution

This has two forms, `$(`*process*`)` and `` ``*process*`` `. They function identically; the first form has two advantages: substitutions can be nested, since the end character is different from the start character, and (because it uses a '`$`') it reminds you that, like parameter substitutions, command substitutions can take place inside double-quoted strings. In that case, like most other things in quotes, the result will be a single word; otherwise, the result is split into words on any field separators you have defined, usually whitespace or the null character. I'll use the `args` function again:

```
% args() { print $# $*; }
% args $(echo two words)
2 two words
% args "$(echo one word)"
1 one word
```

The first form will split on newlines, not just spaces, so an equivalent is

```
% args $(echo two; echo words)
2 two words
```

Thus entire screens of text will be flattened out into a single line of single-word command arguments. By contrast, with the double quotes no processing is done whatsoever; the entire output is put verbatim into one command argument, with newlines intact. This means that the quite common case of wanting a single complete line from a file per command argument has to be handled by trickery; zsh has such trickery, but that's the stuff of chapter 5.

Note the difference from process substitution: no intermediate file name is involved, the output itself goes straight onto the command line. This form of substitution is considerably more common, and, unlike the other, is available in all UNIX shells, though not in all shells with the more modern form '`$(...)`'.

The rule that the command line is evaluated only once, left to right, is adhered to here, but it's a little more complicated in this case since the expression being substituted is scanned *as a complete command line*, so can include anything a command usually can, with all the rules of quoting and expansion being applied. So if you get confused about what a command substitution is actually up to, you should extract the commands from it and think of them as a command line in their own right. When you've worked out what that's doing, decide what it's output will be, and that's the result of the substitution. You can ignore any error output; that isn't captured, so will go straight to the terminal. If you want to ignore it, use the standard trick (see below) '`2>/dev/null`' *inside* the command substitution — not on the main command line, where it won't work because substitutions are performed before redirection of the main command line, and in any case that will have the obvious side effect of changing the error output from the command line itself.

The only real catch with command substitution is that, as it is run as a separate process — even if it only involves shell builtins — no effects other than the output will percolate back to the main shell:

```
% print $(bar=value; print bar is $bar)
bar is value
% print bar is $bar
bar is
```

There is maybe room for a form of substitution that runs inside the shell, instead; however, with modern computers the overhead in starting the extra process is pretty small — and in any case we seem to have run out of new forms of syntax.

Once you know and are comfortable with command substitution, you will probably start using it all the time, so there is one good habit to get into straight away. A particularly common use is simply to put the contents of a file onto the command line.

```
# Don't do this, do the other.
process_cmd `cat file_arguments`
```

But there's a shortcut.

```
# Do do this, don't do the other
process_cmd $(<file_arguments)
```

It's not only less writing, it's more efficient: zsh spots the special syntax, with the `<` immediately inside the parentheses, reads the file directly without bothering to start '`cat`', and inserts its contents: no external process is involved. You shouldn't confuse this with 'null redirections' as described below: the syntax is awfully similar, unfortunately, but the feature shown here is not dependent on that other feature being enabled or set up in a particular way. In fact, this feature works in ksh, which doesn't have zsh's null redirections.

You can quote the file-reading form too, of course: in that case, the contents of the file '`cmd_arguments`' would be passed as just one argument, with newlines and spaces intact.

Sometimes, the rule about splitting the result of a command substitution can get you into trouble:

```
% typeset foo=`echo words words`
% print $foo
words
```

You probably expected the command substitution *not* to be split here. but it was, and the shell executed typeset with the arguments '`foo=words`' and 'words'. That's because in zsh arguments to `typeset` are treated pretty much normally, except for some jiggery pokery with tildes described below. Other shells do this differently, and zsh (from 4.0.2 and 4.1.1) provides a compatibility option, `KSH_TYPESET`. In earlier versions you need to use quotes:

```
% typeset foo="`echo words words`"
% print $foo
words words
```

A really rather technical afterword: using '`$(cat file_arguments)`', you might have counted two extra processes to be started, one being the usual one for a command substitution, and another the '`cat`' process, since that's an external command itself. That would indeed be the obvious way of doing it, but in fact zsh has an optimisation in cases like this: if it knows the shell is about to exit — in this case, the forked process which is just interpreting the command line for the substitution — it will not bother to start a new process for the last command, and here just replaces itself with the `cat`. So actually there's only one extra process here. Obviously, an interactive shell is never replaced in this way, since clairvoyance is not yet a feature of the shell.

Arithmetic substitution

Arithmetic substitution is easy to explain: everything I told you about the `(( ... ))` command under numerical parameters, above, applies to arithmetic substitution. You simply bang a '`$`' in front, and it becomes an expansion.

```
% print $(( 32 + 2 * 5 ))
42
```

You can perform everything inside arithmetic substitution that you can inside the builtin, including assignments; the only difference is that the status is not set, instead the value is put directly onto the command line in place of the original expression. As in C, the value of an assignment is the value being assigned, '`$(( param = 3 + 2))`' substitutes the value 5 as well as assigning it to `$param`.

By the way, there's an extra level of substitution involved in all arithmetic expansions, since scalar parameters are subject to arithmetic expansion when they're read in. This is simple if they only contain numbers, but less obvious if they contain complete expressions:

```
% foo=3+5
% print $(( foo + 2))
10
```

The foo was evaluated into 8 before it was substituted in. Note this means there were two evaluations: this doesn't work:

```
% foo=3+
% print $(( foo 2 ))
zsh: bad math expression: operand expected at ''
```

— the complaint here is about the missing operand after the '+' in the `$foo`. However the following *does* work:

```
% foo=3+
% print $(( $foo 2 ))
5
```

That's because the scalar `$foo` is turned into `3+` first. This is more logical than you might think: with the rule about left to right evaluation, the `$foo` is picked up inside the `$((...))` and expanded as an ordinary parameter substitution while the argument of `$((...))` is being scanned. Then the complete argument '`3+ 2`' is expanded as an arithmetical expression. (Unfortunately, zsh isn't always this logical; there could easily be cases where we haven't thought it through — you should feel free to bring these to our attention.)

There's an older form with single square brackets instead of double parentheses; there is now no reason to use it, as it's non-standard, but you may sometimes still meet it.

Brace expansion

Brace expansion is a feature acquired from the C shell and it's relatives, although some versions of ksh have it, as it's a compile time option there. It's a useful way of saving you

from typing the same thing twice on a single command line:

```
% print -l {foo,bar}' is used far too often in examples'
foo is used far too often in examples
bar is used far too often in examples
```

'print' is given two arguments which it is told to print out one per line. The text in quotes is common to both, but one has 'foo' in front, while the other has 'bar' in front. The brace expression can equally be in the middle of an argument: for example, a common use of this among programmers is for similarly named source files:

```
% print zle_{tricky,vi,word}.c
zle_tricky.c zle_vi.c zle_word.c
```

As you see, you're not limited to two; you can have any number. You can quote a comma if you need a real one:

```
% print -l \`{\,,.}\'' is a punctuation character'
`,' is a punctuation character
`.' is a punctuation character
```

The quotes needed quoting with a backslash to get them into the output. The second comma is the active one for the braces.

You can nest braces. Once again, this is done left to right. In

```
print {now,th{en,ere{,abouts}}}
```

the first argument of the outer brace is 'now', and the second is 'th{en,ere{,abouts}}'. This brace expands to 'then' and then the expansion of 'there{,abouts}', which is 'there thereabouts' — there's nothing to stop you having an empty argument. Putting this all together, we have

```
print now then there thereabouts
```

There's more to know about brace expansion, which will appear in chapter 5 on clever expansions.

3.6.4 Filename Expansion

It's a shame the names 'filename expansion' and 'filename generation' sound so similar, but most people just refer to '~ and = expansion' and 'globbing' respectively, which is all that is meant by the two. The first is by far the simpler. The rule is: unquoted '~'s at the beginning of words perform expansion of named directories, which may be your home directory:

```
% print ~
/home/pws
```

some user's home directory:

```
% print ~root
/root
```

(that may turn up '/' on your system), a directory named directly by you:

```
% t=/tmp
% print ~t
/tmp
```

a directory you've recently visited:

```
% pwd
/home/pws/zsh/projects/zshguide
% print ~+
/home/pws/zsh/projects/zshguide
% cd /tmp
% print ~-
/home/pws/zsh/projects/zshguide
```

or a directory in your directory stack:

```
% pushd /tmp
% pushd ~
% pushd /var/tmp
% print ~2
/tmp
```

These forms were discussed above. There are various extra rules. You can add a '/' after any of them, and the expansions still take place, so you can use them to specify just the first part of a longer expression (as you almost certainly have done with a simple '~'). If you quote the '~' in any of the ways quoting normally takes place, the expansion doesn't happen.

A ~ in the middle of the word means something completely different, if you have the `EXTENDED_GLOB` option set; if you don't, it doesn't mean anything. There are a few exceptions here; assignments are a fairly natural one:

```
% foo=~pws
% print $foo
/home/pws
```

(note that the '`~pws`', being unquoted, was expanded straight away at the assignment, not at the print statement). But the following works too:

```
% PATH=$PATH:~pws/bin
```

because colons are special in assignments. Note that this happens even if the variable isn't a colon-separated path; the shell doesn't know what use you're going to make of all the different variables.

The companion of '~' is '=', which again has to occur at the start of a word or assignment to be special. The remainder of the word (here the *entire* remainder, because directory paths aren't useful) is taken as the name of an external command, and the word is expanded to the complete path to that command, using `$PATH` just as if the command were to be executed:

```
% print =ls
/bin/ls
```

and, slightly confusingly,

```
% foo==ls
% print $foo
/bin/ls
```

where the two '='s have two different meanings. This form is useful in a number of cases. For example, you might want to look at or edit a script which you know is in your path; the form

```
% vi =scriptname
```

is more convenient than the more traditional

```
% vi `whence -p ls`
```

where I put the '-p' in to force `whence` to follow the path, ignoring builtins, functions, etc. This brings us to another use for '=' expansion,

```
% =ls
```

is a neat and extremely short way of referring to an external command when `ls` is usually a function. It has some of the same effect as '`command ls`', but is easier to type.

In versions up to and including `4.0`, this syntax will also expand aliases, so you need to be a bit careful if you really want a path to an external command:

```
% alias foo='ls -F'
% print =foo
ls -F
```

(Path expansion is done in preference, so you are safe if you use `ls`, unless your `$PATH` is strange.) Putting '`=foo`' at the start of the command line doesn't work, and the reason why bears examination: =-expansion occurs quite late on, after ordinary alias expansion and word splitting, so that the result is the single word '`ls -F`', where the space is part of the word, which probably doesn't mean anything (and if it does, don't lend me your computer when I need something done in a hurry). It's probably already obvious that alias expansion here is more trouble than it's worth. A less-than-exhaustive search failed to find anyone who liked this feature, and it has been removed from the shell from 4.1, so that '='-expansion now only expands paths to external commands.

If you don't like =-expansion, you can turn it off by setting the option `NO_EQUALS`. One catch, which might make you want to do that, is that the commands `mmv`, `mcp` and `mln`, which are a commonly used though non-standard piece of free software, use '=' followed by a number to replace a pattern, for example

```
mmv '*.c' '=1.old.c'
```

renames all files ending with `.c` to end with `.old.c`. If you were not alert, you might forget to quote the second word. Otherwise, however, =' isn't very common at the start of a word, so you're probably fairly safe. For a way to do that with zsh patterns, see the discussion of the function `zmv` below (the answer is '`zmv '(*).c' '$1.old.c'`').

Note that zsh is smart enough to complete the names of commands after an '=' of the expandable sort when you hit TAB.

3.6.5 Filename Generation

Filename generation is exactly the same as 'globbing': the expanding of any unquoted wildcards to match files. This is only done in one directory at a time. So for example

```
print *.c
```

won't match files in a subdirectory ending in '`.c`'. However, it *is* done on all parts of a path, so

```
print */*.c
```

will match all '`.c`' files in all immediate subdirectories of the current directory. Furthermore, zsh has an extension — one of its most commonly used special features — to match files in any subdirectory at any depth, including the current directory: use two '`*`'s as part of the path:

```
print **/*.c
```

will match '`prog.c`', '`version1/prog.c`', '`version2/test/prog.c`', '`oldversion/working/saved/prog.c`', and so on. I will talk about filename generation and other uses of zsh's extremely powerful patterns at much greater length in chapter 5. My main thrust here is to fit it into other forms of expansion; the main thing to remember is that it comes last, after everything has already been done.

So although you would certainly expect this to work,

```
print ~/*
```

generating all files in your home directory, you now know why: it is first expanded to '`/home/pws/*`' (or wherever), then the shell scans down the path until it finds a pattern, and looks in the directory it has reached (`/home/pws`) for matching files. Furthermore,

```
foo=~/
print $foo*
```

works. However, as I explained in the last chapter, you need to be careful with

```
foo=*
print ~/$foo
```

This just prints '/home/pws/*'. To get the '*' from the parameter to be a wildcard, you need to tell the shell explicitly that's what you want:

```
foo=*
print ~/${~foo}
```

As also noted, other shells do expand the * as a wildcard anyway. The zsh attitude here, as with word splitting, is that parameters should do exactly what they're told rather than waltz off generating extra words or expansions.

Be even more careful with arrays:

```
foo=(*)
```

will expand the * immediately, in the current directory — the elements of the array assignment are expanded exactly like a normal command line glob. This is often very useful, but note the difference from scalar assignments, which do other forms of expansion, but not globbing.

I'll mention a few possible traps for the unwary, which might confuse you until you are a zsh globbing guru. Firstly, parentheses actually have two uses. Consider:

```
print (foo|bar)(.)
```

The first set of parentheses means 'match either foo or bar'. If you've used egrep, you will probably be familiar with this. The second, however, simply means 'match only regular files'. The '(.)' is called a 'globbing qualifier', because it limits the scope of any matches so far found. For example, if either or both of foo and bar were found, but were directories, they would not now be matched. There are many other possibilities for globbing qualifiers. For now, the easiest way to tell if something at the end is *not* a globbing qualifier is if it contains a '|'.

The second point is about forms like this:

```
print file-<1-10>.dat
```

The '<' and '>' smell of redirection, as described next, but actually the form '<', optional start number, '–', optional finish number, '>' means match any positive integer in the range between the two numbers, inclusive; if either is omitted, there is no limit on that end, hence the cryptic but common '<->' to match any positive integer — in other words, any group of decimal digits (bases other than ten are not handled by this notation). Older versions of the shell allowed the form '<>' as a shorthand to match any number, but the overlap with redirection was too great, as you'll see, so this doesn't work any more.

Another two cryptic symbols are the two that do negation. These only work with the option 'EXTENDED_GLOB' set: this is necessary to get the most out of zsh's patterns, but it can be a trap for the unwary by turning otherwise innocuous characters into patterns:

```
print ^foo
```

This means any file in the current directory *except* the file foo. One way of coming unstuck with '^' is something like

```
stty kill ^u
```

where you would hope '^u' means control with 'u', i.e. ASCII character 21. But it doesn't, if EXTENDED_GLOB is set: it means 'any file in the current directory except one called 'u' ', which is definitely a different thing. The other negation operator isn't usually so fraught, but it can look confusing:

```
print *.c~f*
```

is a pattern of two halves; the shell tries to match '*.c', but rejects any matches which also match 'f*'. Luckily, a '~' right at the end isn't special, so

```
rm *.c~
```

removes all files ending in '.c~' — it wouldn't be very nice if it matched all files ending in '.c' and treated the final '~' as an instruction not to reject any, so it doesn't. The most likely case I can think of where you might have problems is with Emacs' numeric backup files, which can have a '~' in the middle which you should quote. There is no confusion with the directory use of '~', however: that only occurs at the beginning of a word, and this use only occurs in the middle.

The final oddments that don't fit into normal shell globbing are forms with '#'. These also require that EXTENDED_GLOB be set. In the simplest use, a '#' after a pattern says 'match this zero or more times'. So '(foo|bar)#.c' matches foo.c, bar.c, foofoo.c, barbar.c, foobarfoo.c, ... With an extra #, the pattern before (or single character, if it has no special meaning) must match at least once. The other use of '#' is in a facility called 'globbing flags', which look like '(#X)' where 'X' is some letter, possibly followed by digits. These turn on special features from that point in the pattern and are one of the newest features of zsh patterns; they will receive much more space in chapter 5.

3.7 Redirection: greater-thans and less-thans

Redirection means retrieving input from some other file than the usual one, or sending output to some other file than the usual one. The simplest examples of these are '<' and '>', respectively.

```
% echo 'This is an announcement' >tempfile
% cat <tempfile >newfile
% cat newfile
This is an announcement
```

Here, echo sends its output to the file tempfile; cat took its input from that file and sent its output — the same as its input — to the file newfile; the second cat takes its input from newfile and, since its output wasn't redirected, it appeared on the terminal.

The other basic form of redirection is a pipe, using '|'. Some people loosely refer to all redirections as pipes, but that's rather confusing. The input and output of a pipe are *both* programmes, unlike the case above where one end was a file. You've seen lots of examples already:

```
echo foo | sed 's/foo/bar/'
```

Here, `echo` sends its output to the programme `sed`, which substitutes foo by bar, and sends its own output to standard output. You can chain together as many pipes as you like; once you've grasped the basic behaviour of a single pipe, it should be obvious how that works:

```
echo foo is a word |
  sed 's/foo/bar/' |
  sed 's/a word/an unword/'
```

runs another `sed` on the output of the first one. (You can actually type it like that, by the way; the shell knows a pipe symbol can't be at the end of a command.) In fact, a single `sed` will suffice:

```
echo foo is a word |
  sed -e 's/foo/bar/' -e 's/a word/an unword/'
```

has the same effect in this case.

Obviously, all three forms of redirection only work if the programme in question expects input from standard input, and sends output to standard output. You can't do:

```
echo 'edit me' | vi
```

to edit input, since `vi` doesn't use the input sent to it; it always deals with files. Most simple UNIX commands can be made to deal with standard input and output, however. This is a big difference from other operating systems, where getting programmes to talk to each other in an automated fashion can be a major headache.

3.7.1 Clobber

The word 'clobber', as in the option `NO_CLOBBER` which I mentioned in the previous chapter, may be unfamiliar to people who don't use English as their first language. Its basic meaning is 'hit' or 'defeat' or 'destroy', as in 'Itchy and Scratchy clobbered each other with mallets'. If you do:

```
% echo first go >file
% echo second go >file
```

then `file` will contain only the words 'second go'. The first thing you put into the file, 'first go', has been clobbered. Hence the `NO_CLOBBER` option: if this is set, the shell will complain when you try to overwrite the file. You can use '`>|file`' or '`>! file`' to override this. You usually can't use '`>!file`' because history expansion will try to expand '`!file`' before the shell parses the line; hence the form with the vertical bar tends to be more useful.

3.7.2 File descriptors

UNIX-like systems refer to different channels such as input, output and error by 'file descriptors', which are small integers. Usually three are special: 0, standard input; 1,

standard output; and 2, standard error. Bourne-like shells (but not csh-like shells) allow you to refer to a particular file descriptor, instead of standard input or output, by putting the integer immediately before the '<' or '>' (no space is allowed). What's more, if the '<' or '>' is followed immediately by '&', a file descriptor can follow the redirection (the one before is optional as usual). A common use is:

```
% echo This message will go to standard error >&2
```

The command sends its message to standard output, file descriptor 1. As usual, '>' redirects standard output. This time, however, it is redirected not to a file, but to file descriptor 2, which is standard error. Normally this is the same device as standard output, but it can be redirected completely separately. So:

```
% { echo A message
cursh> echo An error >&2 } >file
An error
% cat file
A message
```

Apologies for the slightly unclear use of the continuation prompt 'cursh>': this guide goes into a lot of different formats, and some are a bit finicky about long lines in preformatted text. As pointed out above, the '>file' here will redirect all output from the stuff in braces, just as if it were a single command. However, the '>&2' inside redirects the output of the second echo to standard error. Since this wasn't redirected, it goes straight to the terminal.

Note the form in braces in the previous example — I'm going to use that in a few more examples. It simply sends something to standard output, and something else to standard error; that's its only use. Apart from that, you can treat the bit in braces as a black box — anything which can produce both sorts of output.

Sometimes you want to redirect both at once. The standard Bourne-like way of doing this is:

```
% { echo A message
cursh> echo An error >&2 } >file 2>&1
```

The '>file' redirects standard output from the {...} to the file; the following 2>&1 redirects standard error to wherever standard output happens to be at that point, which is the same file. This allows you to copy two file descriptors to the same place. Note that the order is important; if you swapped the two around, '2>&1' would copy standard error to the initial destination of standard output, which is the terminal, before it got around to redirecting standard output.

Zsh has a shorthand for this borrowed from csh-like shells:

```
% { echo A message
cursh> echo An error >&2 } >&file
```

is exactly equivalent to the form in the previous paragraph, copying standard output and standard error to the same file. There is obviously a clash of syntax with the descriptor-copying mechanism, but if you don't have files whose names are numbers you won't run into it. Note that csh-like shells don't have the descriptor-copying mechanism: the simple '>&' and the same thing with pipes are the only uses of '&' for redirections, and it's not possible there to refer to particular file descriptors.

To copy standard error to a pipe, there are also two forms:

```
% { echo A message
cursh> echo An error >&2 } 2>&1 | sed -e 's/A/I/'
I message
In error
% { echo A message
cursh> echo An error >&2 } |& sed -e 's/A/I/'
I message
In error
```

In the first case, note that the pipe is opened before the other redirection, so that '`2>&1`' copies standard error to the pipe, not the original standard output; you couldn't put that after the pipe in any case, since it would refer to the '`sed`' command's output. The second way is like csh; unfortunately, '`|&`' has a different meaning in ksh (start a coprocess), so zsh is incompatible with ksh in this respect.

You can also close a file descriptor you don't need: the form '`2<&-`' will close standard error for the command where it appears.

One thing not always appreciated about redirections is that they can occur anywhere on the command line, not just at the end.

```
% >file echo foo
% cat file
foo
```

3.7.3 Appending, here documents, here strings, read write

There are various other forms which use multiple '>'s and '<'s. First,

```
% echo foo >file
% echo bar >>file
% cat file
foo
bar
```

The '>>' appends to the file instead of overwriting it. Note, however, that if you use this a lot you may find there are neater ways of doing the same thing. In this example,

```
% { echo foo
cursh> echo bar } >file
% cat file
foo
bar
```

Here, '`cursh>`' is a prompt from the shell that it is waiting for you to close the '`{`' construct which executes a set of commands in the current shell. This construct can have a redirection applied to the entire sequence of commands: '`>file`' after the closing brace therefore redirects the output from both `echo`s.

In the case of input, doubling the sign has a totally different effect. The word after the << is not a file, but a string which will be used to mark in the end of input. Input is read until a line with only this string is found:

```
% sed -e 's/foo/bar/' <<HERE
heredoc> This line has foo in it.
heredoc> There is another foo in this one.
heredoc> HERE
This line has a bar in it.
There is another bar in this one.
```

The shell prompts you with 'heredoc>' to tell you it is reading a 'here document', which is how this feature is referred to. When it finds the final string, in this case 'HERE', it passes everything you have typed as input to the command as if it came from a file. The command in this case is the stream editor, which has been told to replace the first 'foo' on each line with a 'bar'. (Replacing things with a bar is a familiar experience from the city centre of my home town, Newcastle upon Tyne.)

So far, the features are standard in Bourne-like shells, but zsh has an extension to here documents, sometimes referred to as 'here strings'.

```
% sed -e 's/string/nonsense/' \
> <<<'This string is the entire document.'
This nonsense is the entire document.
```

Note that '>' on the second line is a continuation prompt, not part of the command line; it was just too long for the TeX version of this document if I didn't split it. This is a shorthand form of 'here' document if you just want to pass a single string to standard input.

The final form uses both symbols: '<>file' opens the file for reading and writing — but only on standard input. In other words, a programme can now both read from and write to standard input. This isn't used all that often, and when you do use it you should remember that you need to open standard output explicitly to the same file:

```
% echo test >/tmp/redirtest
% sed 's/e/Z/g' <>/tmp/redirtest 1>&0
% cat /tmp/redirtest
tZtst
```

As standard input (the 0) was opened for writing, you can perform the unusual trick of copying standard output (the 1) into it. This is generally not a particularly safe way of doing in-place editing, however, though it seems to work fine with sed. Note that in older versions of zsh, '<>' was equivalent to '<->', which is a pattern that matches any number; this was changed quite some time ago.

3.7.4 Clever tricks: exec and other file descriptors

All Bourne-like shells have two other features. First, the 'command' exec, which I described above as being used to replace the shell with the command you give after it, can be used with only redirections after it. These redirections then apply permanently to the shell itself, rather than temporarily to a single command. So

```
exec >file
```

makes `file` the destination for standard output from that point on. This is most useful in scripts, where it's quite common to want to change the destination of all output.

The second feature is that you can use file descriptors which haven't even been opened yet, as long as they are single digits — in other words, you can use numbers 3 to 9 for your own purposes. This can be combined with the previous feature for some quite clever effects:

```
exec 3>&1
# 3 refers to stdout
exec >file
# stdout goes to 'file', 3 untouched
    # random commands output to 'file'
exec 1>&3
# stdout is now back where it was
exec 3>&-
# file descriptor 3 closed to tidy up
```

Here, file descriptor 3 has been used simply as a placeholder to remember where standard output was while we temporarily divert it. This is an alternative to the '`{...} >file`' trick. Note that you can put more than one redirection on the `exec` line: '`exec 3>&1 >file`' also works, as long as you keep the order the same.

3.7.5 Multios

Multios allow you to do an implicit '`cat`' (concatenate files) on input and '`tee`' (send the same data to different files) on output. They depend on the option `MULTIOS` being set, which it is by default. I described this in the last chapter in discussing whether or not you should have the option set, so you can look at the examples there.

Here's one fact I didn't mention. You use output multios like this:

```
command-generating-output >file1 >file2
```

where the command's output is copied to both files. This is done by a process forked off by the shell: it simply sits waiting for input, then copies it to all the files in its list. There's a problem in all versions of the shell to date (currently 4.0.6): this process is asynchronous, so you can't rely on it having finished when the shell starts executing the next command. In other words, if you look at `file1` or `file2` immediately after the command has finished, they may not yet contain all the output because the forked process hasn't finished writing to it.

This is really a bug, but for the time being you will have to live with it as it's quite complicated to fix in all cases. Multios are most useful as a shorthand in interactive use, like so much of zsh; in a script or function it is safer to use `tee`,

```
command-generating-output | tee file1 file2
```

which does the same thing, but as `tee` is handled as a synchronous process `file1` and `file2` are guaranteed to be complete when the pipeline exits.

3.8 Shell syntax: loops, (sub)shells and so on

3.8.1 Logical command connectors

I have been rather cavalier in using a couple of elements of syntax without explaining them:

```
true  &&  print Previous command returned true
false  ||  print Previous command returned false
```

The relationship between '`&&`' and '`||`' and tests is fairly obvious, but in this case they connect complete commands, not test arguments. The '`&&`' executes the following command if the one before succeeded, and the '`||`' executes the following command if the one before failed. In other words, the first is equivalent to

```
if true; then
  print Previous command returned true
fi
```

but is more compact.

There is a perennial argument about whether to use these or not. In the comp.unix.shell newsgroup on Usenet, you see people arguing that the '`&&`' syntax is unreadable, and only an idiot would use it, while other people argue that the full '`if`' syntax is slower and clumsier, and only an idiot would use that for a simple test; but Usenet is like that, and both answers are a bit simplistic. On the one hand, the difference in speed between the two forms is minute, probably measurable in microseconds rather than milliseconds on a modern computer; the scheduling of the shell process running the script by the operating system is likely to make more difference if these are embedded inside a much longer script or function, as they will be. And on the other hand, the connection between '`&&`' and a logical 'and' is so strong in the minds of many programmers that to anyone with moderate shell experience they are perfectly readable. So it's up to you. I find I use the '`&&`' and '`||`' forms for a pair of simple commands, but use '`if`' for anything more complicated.

I would certainly advise you to avoid chains like:

```
true || print foo && print bar || false
```

If you try that, you will see '`bar`' but not '`foo`', which is not what a C programmer might expect. Using the usual rules of precedence, you would parse it as: either `true` must be true; or both the `print` statements must be true; or the false must be true. However, the shell parses it differently, using these rules:

- If you encounter an '`&&`',
  - if the command before it (really the complete pipeline) succeeded, execute the command immediately after, and execute what follows normally
  - else if the command failed, skip the next command and any others until an '`||`' is encountered, or until the group of commands is ended by a newline, a semicolon, or the end of an enclosing group. Then execute whatever follows in the normal way.
- If you encounter an '`||`',

- if the command before it succeeded, skip the next command and any others until an '`&&`' is encountered, or until the end of the group, and execute what follows normally
- else if the command failed, execute the command immediately after the '`||`'.

If that's hard to follow, just note that the rule is completely symmetric; a simple summary is that the logical connectors don't remember their past state. So in the example shown, the '`true`' succeeds, we skip '`print foo`' but execute '`print bar`' and then skip `false`. The expression returns status zero because the last thing it executed did so. Oddly enough, this is completely standard behaviour for shells. This is a roundabout way of saying 'don't use combined chains of '`&&`'s and '`||`'s unless you think Gödel's theorem is for sissies'.

Strictly speaking, the and's and or's come in a hierarchy of things which connect commands. They are above pipelines, which explains my remark above — an expression like '`echo $ZSH_VERSION |sed '/dev//'`' is treated as a single command between any logical connectors — and they are below newlines and semicolons — an expression like '`true && print yes; false ||print no`' is parsed as two distinct sets of logically connected command sequences. In the manual, a list is a complete set of commands executed in one go:

```
echo foo; echo bar

echo small furry animals
```

— a shell function is basically a glorified list with arguments and a name. A sublist is a set of commands up to a newline or semicolon, in other words a complete expression possibly involving the logical connectors:

```
show -nomoreproc |
  grep -q foo &&
  print The word ''`foo'\' occurs.
```

A pipeline is a chain of one or more commands connected by '`|`', for example both individual parts of the previous sublist,

```
show -nomoreproc | grep -q foo
```

and

```
print The word ''`foo'\' occurs.
```

count as pipelines. A simple command is one single unit of execution with a command name, so to use the same example that includes all three of the following,

```
show -nomoreproc
grep -q foo
print The word ''`foo'\' occurs.
```

This means that in something like

```
print foo
```

where the command is terminated by a newline and then executed in one go, the expression is all of the above — list, sublist, pipeline and simple command. Mostly I won't need to make the formal distinction; it sometimes helps when you need to break down a complicated set of commands. It's a good idea, and usually possible, to write in such a way that it's obvious how the commands break down. It's not too important to know the details, as long as you've got a feel for how the shell finds the next command.

3.8.2 Structures

I've shown plenty of examples of one sort of shell structure already, the `if` statement:

```
if [[ black = white ]]; then
  print Yellow is no colour.
fi
```

The main points are: the '`if`' itself is followed by some command whose return status is tested; a '`then`' follows as a new command; any number of commands may follow, as complex as you like; the whole sequence is ended by a '`fi`' as a command on its own. You can write the '`then`' on a new line if you like, I just happen to find it neater to stick it where it is. If you follow the form here, remember the semicolon before it; the `then` must start a separate command. (You can put another command immediately after the `then` without a newline or semicolon, though, although people tend not to.)

The double-bracketed test is by far the most common thing to put here in zsh, as in ksh, but any command will do; only the status is important.

```
if true; then
  print This always gets executed
fi
if false; then
  print This never gets executed
fi
```

Here, `true` always returns true (status 0), while `false` always returns false (status 1 in zsh, although some versions return status 255 — anything nonzero will do). So the statements following the `print`s are correct.

The `if` construct can be extended by '`elif`' and '`else`':

```
read var
if [[ $var = yes ]]; then
  print Read yes
elif [[ $var = no ]]; then
  print Read no
else
  print Read something else
fi
```

The extension is pretty straightforward. You can have as many '`elif`'s with different tests as you like; the code following the first test to succeed is executed. If no test succeeded, and there is an '`else`' (there doesn't need to be), the code following that is executed. Note that the

form of the 'elif' is identical to that of 'if', including the 'then', while the else just appears on its own.

The while-loop is quite similar to if. There are two differences: the syntax uses while, do and done instead of if, then and fi, and after the loop body is executed (if it is), the test is evaluated again. The process stops as soon as the test is false. So

```
i=0
while (( i++ < 3 )); do
  print $i
done
```

prints 1, then 2, then 3. As with if, the commands in the middle can be any set of zsh commands, so

```
i=0
while (( i++ < 3 )); do
  if (( i & 1 )); then
    print $i is odd
  else
    print $i is even
  fi
done
```

tells you that 1 and 3 are odd while 2 is even. Remember that the indentation is irrelevant; it is purely there to make the structures more easy to understand. You can write the code on a single line by replacing all the newlines with semicolons.

There is also an until loop, which is identical to the while loop except that the loop is executed until the test is true. 'until [[...' is equivalent to 'while ! [[...'.

Next comes the for loop. The normal case can best be demonstrated by another example:

```
for f in one two three; do
  print $f
done
```

which prints out 'one' on the first iteration, then 'two', then 'three'. The f is set to each of the three words in turn, and the body of the loop executed for each. It is very useful that the words after the 'in' may be anything you would normally have on a shell command line. So 'for f in *; do' will execute the body of the loop once for each file in the current directory, with the file available as $f, and you can use arrays or command substitutions or any other kind of substitution to generate the words to loop over.

The for loop is so useful that the shell allows a shorthand that you can use on the command line: try

```
for f in *; print $f
```

and you will see the files in the current directory printed out, one per line. This form, without the do and the done, involves less typing, but is also less clear, so it is recommended that you only use it interactively, not in scripts or functions. You can turn the feature off with NO_SHORT_LOOPS.

The `case` statement is used to test a pattern against a series of possibilities until one succeeds. It is really a short way of doing a series of `if` and `elif` tests on the same pattern:

```
read var
case $var in
  (yes) print Read yes
        ;;
  (no) print Read no
       ;;
  (*) print Read something else
      ;;
 esac
```

is identical to the `if`/`elif`/`else` example above. The `$var` is compared against each pattern in turn; if one matches, the code following that is executed — then the statement is exited; no further matches are looked for. Hence the '`*`' at the end, which can match anything, acts like the '`else`' of an `if` statement.

Note the quirks of the syntax: the pattern to test must appear in parentheses. For historical reasons, you can miss out the left parenthesis before the pattern. I haven't done that mainly because unbalanced parentheses confuse the system I am using for writing this guide. Also, note the double semicolon: this is the only use of double semicolons in the shell. That explains the fact that if you type '`;;`' on its own the shell will report a 'parse error'; it couldn't find a `case` to associate it with.

You can also use alternative patterns by separating them with a vertical bar. Zsh allows alternatives with extended globbing anyway; but this is actually a separate feature, which is present in other shells which don't have zsh's extended globbing feature; it doesn't depend on the `EXTENDED_GLOB` option:

```
read var
case $var in
  (yes|true|1) print Reply was affirmative
               ;;
  (no|false|0) print Reply was negative
               ;;
  (*) print Reply was cobblers
           ;;
esac
```

The first '`print`' is used if the value of `$var` read in was '`yes`', '`true`' or '`1`', and so on. Each of the separate items can be a pattern, with any of the special characters allowed by zsh, this time depending on the setting of the option `EXTENDED_GLOB`.

The `select` loop is not used all that often, in my experience. It is only useful with interactive input (though the code may certainly appear in a script or function):

```
select var in earth air fire water; do
  print You selected $var
done
```

This prints a menu; you must type 1, 2, 3 or 4 to select the corresponding item; then the body of the loop is executed with `$var` set to the value in the list corresponding to the number.

To exit the loop hit the break key (usually `^G`) or end of file (usually `^D`: the feature is so infrequently used that currently there is a bug in the shell that this tells you to use '`exit`' to exit, which is nonsense). If the user entered a bogus value, then the loop is executed with `$var` set to the empty string, though the actual input can be retrieved from `$REPLY`. Note that the prompt printed for the user input is `$PROMPT3`, the only use of this parameter in the shell: all normal prompt substitutions are available.

There is one final type of loop which is special to zsh, unlike the others above. This is '`repeat`'. It can be used two ways:

```
% repeat 3 print Hip Hip Hooray
Hip Hip Hooray
Hip Hip Hooray
Hip Hip Hooray
```

Here, the first word after `repeat` is a count, which could be a variable as normal substitutions are performed. The rest of the line (or until the first semicolon) is a command to repeat; it is executed identically each time.

The second form is a fully fledged loop, just like `while`:

```
% repeat 3; do
repeat> print Hip Hip Hooray
repeat> done
Hip Hip Hooray
Hip Hip Hooray
Hip Hip Hooray
```

which has the identical effect to the previous one. The '`repeat>`' is the shell's prompt to show you that it is parsing the contents of a '`repeat`' loop.

3.8.3 Subshells and current shell constructs

More catching up with stuff you've already seen. The expression in parentheses here:

```
% (cd ~; ls)
<all the files in my home directory>
% pwd
<where I was before, not necessarily ~>
```

is run in a subshell, as if it were a script. The main difference is that the shell inherits almost everything from the main shell in which you are typing, including options settings, functions and parameters. The most important thing it doesn't inherit is probably information about jobs: if you run `jobs` in a subshell, you will get no output; you can't use `fg` to resume a job in a subshell; you can't use '`kill %`*n*' to kill a job (though you can still use the process ID); and so on. By now you should have some feel for the effect of running in a separate process. Running a command, or set of commands, in a different directory, as in this example, is one quite common use for this construct. (In zsh 4.1, you can use `jobs` in a subshell; it lists the jobs running in the parent shell; this is because it is very useful to be able to pipe the output of jobs into some processing loop.)

On the other hand, the expression in braces here:

```
% {cd ~; ls}
<all the files in my home directory>
% pwd
/home/pws
```

is run in the current shell. This is what I was blathering on about in the section on redirection. Indeed, unless you need some special effect like redirecting a whole set of commands, you won't use the current-shell construct. The example here would behave just the same way if the braces were missing.

As you might expect, the syntax of the subshell and current-shell forms is very similar. You can use redirection with both, just as with simple commands, and they can appear in most places where a simple command can appear:

```
[[ $test = true ]] && {
  print Hello.
  print Well, this is exciting.
}
```

That would be much clearer using an '`if`', but it works. For some reason, you often find expressions of this form in system start-up files located in the directory `/etc/rc.d` or, on older systems, in files whose names begin with '`/etc/rc.`'. You can even do:

```
if { foo=bar; [[ $foo = bar ]] }; then
  print yes
fi
```

but that's also pretty gross.

One use for `{...}` is to make sure a whole set of commands is executed at once. For example, if you copy a set of commands from a script in one window and want them to be run in one go in a shell in another window, you can do:

```
% {
cursh>            # now paste your commands in here...
 ...
cursh> }
```

and the commands will only be executed when you hit return after the final '`}`'. This is also a workaround for some systems where cut and paste has slightly odd effects due to the way different states of the terminal are handled. The current-shell construct is a little bit like an anonymous function, although it doesn't have any of the usual features of functions — you can't pass it arguments, and variables declared inside aren't local to that section of code.

3.8.4 Subshells and current shells

In case you're confused about what happens in the current shell and what happens in a subshell, here's a summary.

The following are run in the current shell.

1. All shell builtins and anything which looks like one, such as a precommand modifier and tests with '`[[`'.
2. All complex statements and loops such as `if` and `while`. Tests and code inside the block must both be considered separately.
3. All shell functions.
4. All files run by '`source`' or '`.`' as well as startup files.
5. The code inside a '`{...}`'.
6. The right hand side of a pipeline: this is guaranteed in zsh, but don't rely on it for other shells.
7. All forms of substitution except `` `... ` ``, `$(...)`, `=(...)`, `<(...)` and `>(...)`.

The following are run in a subshell.

1. All external commands.
2. Anything on the left of a pipe, i.e. all sections of a pipeline but the last.
3. The code inside a '(...)'.
4. Substitutions involving execution of code, i.e. `` `... ` ``, `$(...)`, `=(...)`, `<(...)` and `>(...)`. (TCL fans note that this is different from the '`[...]`' command substitution in that language.)
5. Anything started in the background with '`&`' at the end.
6. Anything which has ever been suspended. This is a little subtle: suppose you execute a set of commands in the current shell and suspend it with `^Z`. Since the shell needs to return you to the prompt, it forks a subshell to remember the commands it was executing when you interrupted it. If you use `fg` or `bg` to restart, the commands will stay in the subshell. This is a special feature of zsh; most shells won't let you interrupt anything in the current shell like that, though you can still abort it with `^C`.

With an alias, you can't tell where it will be executed — you need to find out what it expands too first. The expansion naturally takes place in the current shell.

Of course, if for some reason the current set of commands is already running in a subshell, it doesn't get magically returned to the current shell — so a shell builtin on the left hand side of a pipeline is running in a subshell. However, it doesn't get an extra subshell, as an external command would. What I mean is:

```
{ print Hello; cat file } |
  while read line; print $line; done
```

The shell forks, producing a subshell, to execute the left hand side of the pipeline, and that subshell forks to execute the `cat` external command, but nothing else in that set of commands will cause a new subshell to be created.

(For the curious only: actually, that's not quite true, and I already pointed this out when I talked about command substitutions: the shell keeps track of occasions when it is in a subshell and has no more commands to execute. In this case it will not bother forking to create a new process for the `cat`, it will simply replace the subshell which is not needed any more. This can only happen in simple cases where the shell has no clearing up to do.)

3.9 Emulation and portability

I described the options you need to set for compatibility with ksh in the previous chapter. Here I'm more interested in the best way of running ksh scripts and functions.

First, you should remember that because of all zsh's options you can't assume that a piece of zsh code will simply run a piece of sh or ksh code without any extra changes. Our old friend `SH_WORD_SPLIT` is the most common problem, but there are plenty of others. In addition to options, there are other differences which simply need to be worked around. I will list some of them a bit later. Generally speaking, Bourne shell is simple enough that zsh emulates it pretty well — although beware in case you are using bash extensions, since to many Linux users bash is the nearest approximation to the Bourne shell they ever come across. Zsh makes no attempt to emulate bash, even though some of bash's features have been incorporated.

To make zsh emulate ksh or sh as closely as it knows how, there are various things you can do.

1. Invoke zsh under the name sh or ksh, as appropriate. You can do this by creating a symbolic link from zsh to sh or ksh. Then when zsh starts up all the options will be set appropriately. If you are starting that shell from another zsh, you can use the feature of zsh that tricks a programme into thinking it has a different name: '`ARGV0=sh zsh`' runs zsh under the name sh, just like the symbolic link method.
2. Use '`emulate ksh`' at the top of the script or function you want to run. In the case of a function, it is better to run '`emulate -L ksh`' since this makes sure the normal options will be restored when the function exits; this is irrelevant for a script as the options cannot be propagated to the process which ran the script. You can also use the option '`-R`' after `emulate`, which forces more options to be like ksh; these extra options are generally for user convenience and not relevant to basic syntax, but in some cases you may want the extra cover provided.

   If it's possible the script may already be running under ksh, you can instead use

   ```
   [[ -z $ZSH_VERSION ]] && emulate ksh
   ```

   or for sh, using the simpler test command there,

   ```
   [ x$ZSH_VERSION = x ] && emulate sh
   ```

Both these methods have drawbacks, and if you plan to be a heavy zsh user there's no substitute for simply getting used to zsh's own basic syntax. If you think there is some useful element of emulation we missed, however, you should certainly tell the zsh-workers mailing list about it.

Emulation of ksh88 is much better than emulation of ksh93. Support for the latter is gradually being added, but only patchily.

There is no easy way of converting code written for any csh-like shell; you will just have to convert it by hand. See the FAQ for some hints on converting aliases to functions.

3.9.1 Differences in detail

Here are some differences from ksh88 which might prove significant for ksh programmers. This is lifted straight from the corresponding section of the FAQ; it is not complete, and

indeed some of the 'differences' could be interpreted as bugs. Those marked '*' perform in a ksh-like manner if the shell is invoked with the name 'ksh', or if 'emulate ksh' is in effect.

- Syntax:
  - * Shell word splitting.
  - * Arrays are (by default) more csh-like than ksh-like: subscripts start at 1, not 0; `array[0]` refers to `array[1]`; `$array` refers to the whole array, not `$array[0]`; braces are unnecessary: `$a[1] == ${a[1]}`, etc. The `KSH_ARRAYS` option is now available.
  - Coprocesses are established by `coproc`; `|&` behaves like csh. Handling of coprocess file descriptors is also different.
  - In `cmd1 && cmd2 &`, only `cmd2` instead of the whole expression is run in the background in zsh. The manual implies this is a bug. Use `{ cmd1 && cmd2 } &` as a workaround.
- Command line substitutions, globbing etc.:
  - * Failure to match a globbing pattern causes an error (use `NO_NOMATCH`).
  - * The results of parameter substitutions are treated as plain text: `foo="*"; print $foo` prints all files in ksh but `*` in zsh (unset `GLOB_SUBST`).
  - * `$PSn` do not do parameter substitution by default (use `PROMPT_SUBST`).
  - * Standard globbing does not allow ksh-style 'pattern-lists'. See chapter 5 for a list of equivalent zsh forms. The `^`, `~` and `#` (but not `|`) forms require `EXTENDED_GLOB`. From version 3.1.3, the ksh forms are fully supported when the option `KSH_GLOB` is in effect.

    [1] Note that `~` is the only globbing operator to have a lower precedence than `/`. For example, `**/foo~*bar*` matches any file in a subdirectory called `foo`, except where `bar` occurred somewhere in the path (e.g. `users/barstaff/foo` will be excluded by the `~` operator). As the `**` operator cannot be grouped (inside parentheses it is treated as `*`), this is the way to exclude some subdirectories from matching a `**`.
  - Unquoted assignments do file expansion after colons (intended for PATHs).
  - `integer` does not allow `-i`.
  - `typeset` and `integer` have special behaviour for assignments in ksh, but not in zsh. For example, this doesn't work in zsh:

    ```
    integer k=$(wc -l ~/.zshrc)
    ```

    because the return value from `wc` includes leading whitespace which causes wordsplitting. Ksh handles the assignment specially as a single word.
- Command execution:
  - * There is no `$ENV` variable (use `/etc/zshrc`, `~/.zshrc`; note also `$ZDOTDIR`).
  - `$PATH` is not searched for commands specified at invocation without -c.
- Aliases and functions:
  - The order in which aliases and functions are defined is significant: function definitions with () expand aliases.
  - Aliases and functions cannot be exported.

  - There are no tracked aliases: command hashing replaces these.
  - The use of aliases for key bindings is replaced by 'bindkey'.
  - * Options are not local to functions (use LOCAL_OPTIONS; note this may always be unset locally to propagate options settings from a function to the calling level).

- Traps and signals:
  - * Traps are not local to functions. The option LOCAL_TRAPS is available from 3.1.6.
  - TRAPERR has become TRAPZERR (this was forced by UNICOS which has SIGERR).

- Editing:
  - The options `emacs`, `gmacs`, `viraw` are not supported. Use bindkey to change the editing behaviour: `set -o {emacs,vi}` becomes `bindkey -{e,v}`; for gmacs, go to emacs mode and use `bindkey \^t gosmacs-transpose-characters`.
  - The `keyword` option does not exist and `-k` is instead interactivecomments. (`keyword` will not be in the next ksh release either.)
  - Management of histories in multiple shells is different: the history list is not saved and restored after each command. The option `SHARE_HISTORY` appeared in 3.1.6 and is set in ksh compatibility mode to remedy this.
  - `\` does not escape editing chars (use `^V`).
  - Not all ksh bindings are set (e.g. `<ESC>#`; try `<ESC>q`).
  - * `#` in an interactive shell is not treated as a comment by default.

- Built-in commands:
  - Some built-ins (`r`, `autoload`, `history`, `integer` ...) were aliases in ksh.
  - There is no built-in command newgrp: use e.g. `alias newgrp="exec newgrp"`
  - `jobs` has no `-n` flag.
  - `read` has no `-s` flag.

- Other idiosyncrasies:
  - `select` always redisplays the list of selections on each loop.

3.9.2 Making your own scripts and functions portable

There are also problems in making your own scripts and functions available to other people, who may have different options set.

In the case of functions, it is always best to put '`emulate -L zsh`' at the top of the function, which will reset the options to the default zsh values, and then set any other necessary options. It doesn't take the shell a great deal of time to process these commands, so try and get into the habit of putting them any function you think may be used by other people. (Completion functions are a special case as the environment is already standardised — see chapter 6 for this.)

The same applies to scripts, since if you run the script without using the option '`-f`' to zsh the user's non-interactive startup files will be run, and in any case the file `/etc/zshenv` will be run. We urge system administrators not to set options unconditionally in that file unless absolutely necessary; but they don't always listen. Hence an `emulate` can still save a lot of grief.

3.10 Running scripts

Here are some final comments on running scripts: they apply regardless of the problems of portability, but you should certainly also be aware of what I was saying in the previous section.

You may be aware that you can force the operating system to run a script using a particular interpreter by putting '`#!`' and the path to the interpreter at the top of the script. For example, a zsh script could start with

```
#!/usr/local/bin/zsh
print The arguments are $*
```

assuming that zsh lives in the directory `/usr/local/bin`. Then you can run the script under its name as if it were an ordinary command. Suppose the script were called '`scriptfile`' and in the current directory, and you want to run it with the arguments '`one two forty-three`'. First you must make sure the script is executable:

```
% chmod +x scriptfile
```

and then you can run it with the arguments:

```
% ./scriptfile one two forty-three
The arguments are one two forty-three
```

The shell treats the first line as a comment, since it begins with a '`#`', but note it still gets evaluated by the shell; the system simply looks inside the file to see if what's there, it doesn't change it just because the first line tells it to execute the shell.

I put the '`./`' in front to refer to the current directory because I don't usually have that in my path — this is for safety, to avoid running things which happen to have names like commands simply because they were in the current directory. But many people aren't so paranoid, and if '`.`' is in your path, you can omit the '`./`'. Hence, obviously, it can be anywhere else in your path: it is searched for as an ordinary executable.

The shell actually provides this mechanism even on operating systems (now few and far between in the UNIX world) that don't have the feature built into them. The way this works is that if the shell found the file, and it was executable, but running it didn't work, then it will look for the `#!`, extract the name following and run (in this example) '`/usr/local/bin/zsh` *<path>*/scriptfile `one two forty-three`', where *<path>* is the path where the file was found. This is, in fact, pretty much what the system does if it handles it itself.

Some shells search for scripts using the path when they are given as filenames at invocation, but zsh happens not to. In other words, '`zsh scriptfile`' only runs `scriptfile` in the current directory.

There are two other features you may want to be aware of. Both are down to the operating system, if that is what is responsible for the '`#!`' trick (true of all the most common UNIX-like systems at the moment). First, you are usually allowed to supply one, but only one, argument or option in the '`#!`' line, thus:

```
#!/usr/local/bin/zsh -f
print other stuff here
```

which stops startup files other than `/etc/zshenv` from being run, but otherwise works the same as before. If you need more options, you should combine them in the same word. However, it's usually clearer, for anything apart from `-f`, `-i` (which forces the shell into interactive mode) and a few other options which need to take effect immediately, to put a '`setopt`' line at the start of the body of the script. In a few versions of zsh, there was an unexpected consequence of the fact that the line would only be split once: if you accidentally left some spaces at the end of the line (e.g. '`#!/usr/local/bin/zsh -f `') they would be passed down to the shell, which would report an error, which was hard to interpret. The spaces will still usually be passed down, but the shell is now smart enough to ignore spaces in an option list.

The second point is that the length of the '`#!`' line which will be evaluated is limited. Often the limit is 32 characters, in total, That means if your path to zsh is long, e.g. '`/home/users/psychology/research/dreams/freud/solaris_2.5/bin/zsh`' the system won't be able to find the shell. Your only recourse is to find a shorter path, or execute the shell directly, or some sneakier trick such as running the script under `/bin/sh` and making that start zsh when it detects that zsh isn't running yet. That's a fairly nasty way of doing it, but just in case you find it necessary, here's an example:

```
#!/bin/sh

if [ x$ZSH_VERSION = x ]; then
  # Put the right path in here ---
  # or just rely on finding zsh in
  # $path, since 'exec' handles that.
  exec /usr/local/bin/zsh $0 "$@"
fi

print $ZSH_VERSION
print Hello, this is $0
print with arguments $*.
```

Note that first '`$0`', which passes down the name of the script that was originally executed. Running this as '`testexec foo bar`' gives me

```
3.1.9-dev-8
Hello, this is /home/pws/tmp/testexec
with arguments foo bar.
```

I hope you won't have to resort to that. By the way, really, excruciatingly old versions of zsh didn't have `$ZSH_VERSION`. Rather than fix the script, I suggest you upgrade the shell. Also, on some old Bourne shells you might need to replace `"$@"` with `${1+"$@"}`, which is more careful about only putting in arguments if there were any (this is the sort of thing we'll see in chapter 5). Usually this isn't necessary.

You can use the same trick on ancient versions of UNIX which didn't handle '`#!`'. On some such systems, anything with a '`:`' as the first character is run with the Bourne shell, so this serves as an alternative to '`#!/bin/sh`', while on some Berkeley systems, a plain '`#`' caused csh to be used. In the second case, you will need to change the syntax of the first test to be understood by both zsh and csh. I'll leave that as an exercise for the reader. If you have perl (very probable these days) you can look at the `perlrun` manual page, which discusses the corresponding problem of starting perl scripts from a shell, for some ideas.

There's one other glitch you may come across. Sometimes if you type the name of a script

which you know is in your path and is executable, the shell may tell you '`file not found`', or some equivalent message. What this usually means is that the *interpreter* wasn't found, because you mistyped the line after the '`#!`'. This confusing message isn't the shell's fault: a lot of operating systems return the same system error in this case as if the script were really not found. It's not worth the shell searching the path to see if the script is there, because in the vast majority of cases the error refers to the programme in the execution path. If the operating system returned the more natural error, '`exec format error`', then the shell would know that there was something wrong with the file, and could investigate; but unfortunately life's not that simple.

Chapter 4

The Z-Shell Line Editor

The zsh line editor is probably the first part of the shell you ever used, when you started typing in commands. Even the most basic shells, such as sh, provide some kind of editing capability, although in that case probably just what the system itself does — enter characters, delete the last character, delete the entire line. Most shells you're likely to use nowadays do quite a lot more. With zsh you can even extend the set of editor commands using shell functions.

4.1 Introducing zle

The zsh line editor is usually abbreviated to 'zle'. Normally it fires itself up for any interative shell; you don't have to do anything special until you decide you need to change its behaviour. If everything looks OK and you're not interested in how zle is started up, skip to the next subsection.

Nowadays, zle lives in its own loadable module, `zsh/zle`, which saves all the overhead of having an editor if the shell isn't interactive. However, you normally won't need to worry about that; I'll say more about modules in chapter 7, but the shell knows when you need zle and gives you it automatically. Usually the module is in a directory with a name like '`/usr/local/lib/zsh/4.0.4/zsh/zle.so`', where the '`4.0.4`' is the shell's version number, the same as the value of the parameter `$ZSH_VERSION`, and everything after that apart from the suffix '`.so`' is the module name. The suffix may be '`.sl`' (HP-UX) or '`.dll`' (Cygwin), but '`.so`' is by far the most common form. It differs because zsh keeps the same convention for dynamically loadable libraries, or 'shared objects' in UNIX-speak, as the operating system.

If the shell is badly installed, you sometimes see error messages that it, or a command such as 'bindkey', couldn't be loaded. That means the shell couldn't find '`zsh/zle`' anywhere in the module load path, the array `$module_path`. Then you need to complain to your system administrator. If you've just compiled zsh and are having this problem, it's because you have to install the modules before you run the shell, even if the shell itself isn't installed. You can do that by saying '`make install.modules`'. Then the compiled zsh should run from where it is.

Note that unlike bash's line editor, readline, which is an entirely separate library, zle is an integral part of the shell. Hence you configure it by sticking commands in your `.zshrc` —

as it's only useful for an interactive shell, only `/etc/zshrc` and `.zshrc` make sense for this purpose.

One tip if you're looking at the zsh manual using info, either with the command of that name or `\C-h i` within Emacs, which I find the most convenient way: the entry for zle is called 'Zsh Line Editor', in full, not just 'Zle'. Have fun looking for 'Shell Builtin Commands' (not 'Builtins') while you're at it.

4.1.1 The simple facts

As with any editor later than `ed`, you can move around the line and change it using various 'keystrokes', in other words one or more sets of keys you type at once. For example, the keystroke to move back a word is (maybe) `ESC b`. This means you first hit the escape key; nothing happens yet. Then you hit 'b', and the cursor instantly jumps back to the start of the word. (I'll have more to say on what zle thinks is a 'word' — it's not necessarily the same as what the rest of the shell thinks is a word.)

It will probably help if I introduce the shell's way of describing keystrokes right away; then when you need to enter them you can just copy them straight in. The escape key is '`\e`', so that keystroke would be '`\eb`'. Other common keystrokes include holding down the control key, probably near the bottom left of the keyboard, and typing another key at the same time. The simplest way of indicate a control key is just to put '`^`' in front; so for example '`^x^x`' means hold down control, and press 'x' twice with control still held down. It has exactly the same effect as '`^X^X`'. (You may find each time you do that it takes you to the start of the line and back to where you were.)

I've already introduced the weasel word 'maybe' to try to avoid lying. This is because actually zle has two modes of operation, one (the default) like Emacs, the other like vi. If you don't know either of those venerable UNIX editors, I suggest you stick to Emacs mode, since it tends to interfere a little less with what you're doing, and furthermore completion is a little easier. Completion is an offshoot of zle behaviour which is described in chapter 6 (which, you will notice, is longer than this one).

If you normally use vi, you may have one or both of the environment variables `$EDITOR` or `$VISUAL` set to '`vi`'. (It all works the same way if you use '`vim`' instead, or any editor that happens to contain '`vi`' such as '`elvis`'.) In that case, zle will start up in its '`vi`' mode, where the keystrokes are rather different. That's why you might have found that '`\eb`' didn't do what I said, even though you had made no attempt to configure zle. You can make zle always use either emacs or vi mode by putting either

```
bindkey -e
```

or

```
bindkey -v
```

in your `.zshrc`. This is just one of many uses of `bindkey`.

If you're not familiar with this use of the word 'bind', it just means 'make a keystroke execute a particular editor command'. Commands have long-winded names with hyphens which give you quite a good description of what they do, such as '`backward-delete-char`'. Normal keys which correspond to printable characters are usually 'bound to `self-insert`', a perverse way of saying they do what you expect and show up the character you typed.

However, you can actually bind them to something else. In vi command mode, this is perfectly normal.

Actually, if you use a windowing system, you might want to say '`bindkey -me`', which binds a whole set of 'meta' keys. In X Windows, one of the keys on your keyboard, possibly `ALT`, may be designated a 'meta' key which has a special effect similar to the control key. Bindings with the meta key held down are described a bit like they are in Emacs, '`\M-b`'. (You can specify control keys similarly, in fact, like '`\C-x`', but '`^x`' is shorter.) Using the '`-m`' option to `bindkey` tells zsh that wherever it binds an escape sequence like '`\eb`', it should also bind the corresponding meta sequence like '`\M-b`'. Emacs always ties these together, but zsh doesn't — you can rebind them separately. and if you want both sequences to be bound to a new command, you have to bind them both explicitly.

You need to be careful with '`bindkey -m`', however; the shell can't tell whether you are typing a character with the top bit set, or executing a command. This is likely to become worse as the UTF-8 encoding for characters becomes more popular, since a non-ASCII character then consists of a whole series of bytes with the top bit set.

If you are interested in binding function keys, you may already have found the key sequences they send apparently don't make any sense; see the section below for more information. This will introduce the function called `zkbd` which can make the process less painful. The function also helps with 'meta' and 'ALT' keys.

4.1.2 Vi mode

I'm going to concentrate on Emacs mode for various reasons: firstly, because I use it myself; secondly, because the most likely reason for you using vi mode is that you are already familiar with vi and don't need to be told how it works; thirdly, because most of the commands are the same in both modes, just bound differently; and finally because if you *don't* already know vi, you will quite likely find vi editing mode rather counterintuitive and difficult to use.

However, here are a few remarks on it just to get it out of the way. Like the real vi editor, there are two basic modes, insert mode, where you type in text, and command mode, where the same keystrokes which insert characters are bound instead to editing commands. Unlike the real vi, the line editor starts in insert mode on every new command you edit. This means you can often simply type a line up to the 'return' at the end and forget you are in vi mode at all.

To enter command mode, you hit 'escape', again just like normal vi. At this point you are in the magic world of vi commands, where typing an ordinary character can have any effect whatsoever. However, the bindings are similar to normal vi, so '`h`' and '`l`' move left and right. When you want to insert more text, you can use any of the normal vi commands which allow you to do that, such as '`i`' (`vi-insert`) or '`a`' (`vi-add-next`).

Apart from the separate command and insert modes and the completely different set of key bindings, there is no basic difference between Emacs mode and vi mode. You can bind keys in both the vi modes — they don't *have* to correspond to `self-insert` in insert mode. Below, I'll describe 'keymaps', a complete set of descriptions for what all the keys will do (less impressive than it sounds, since a lot of keys may be set to '`undefined-key`, which means they don't do anything useful), and you will see how to change the behaviour in both modes.

4.2 Basic editing

If you know Emacs or vi, you will very soon find out how to do simple commands like moving the cursor, going up and down the history list, and deleting and copying words. If you don't, you should read the `zshzle` manual page for a concise description of what it can do. Here is a summary for Emacs mode.

4.2.1 Moving

You can move forwards and backwards along the line using the cursor keys. The are a variety of different conventions as to what keystrokes the cursor keys produce. You might naively expect that pressing, say, cursor right, sends a signal along the lines of 'cursor right' to the application. Unfortunately, there is no such character in the ASCII character set, so programmes which read input as a string of characters like zsh have to be given an arbitrary string of characters. (It's different for programmes which understand other forms of input, like windowing systems.)

The two most common conventions for cursor keys are where the up key sends '`\e[A`' and the other three the same with `B`, `C` and `D` at the end, and the convention where the '`[`' is replaced by an '`O`' (uppercase letter 'O'). In old versions of zsh, the only convention supported was the first of those two. The second, and any other convention, were not supported at all and you had to bind the keys yourself. This was done by something like:

```
bindkey "\eOA" up-line-or-history
bindkey "\eOB" down-line-or-history
bindkey "\eOC" forward-char
bindkey "\eOD" backward-char
```

The shell tries harder now, and provided your system has the correct information about your terminal (zsh uses an old system called 'termcap' which has largely been superseded by another called 'terminfo') you should be lucky. If the shell thinks your keys are too perverse — in particular, if the keystroke it wants to bind the function too is already defined by zsh — you will still have to do it by hand. The list above should serve as a template.

Instead of the cursor keys, traditional Emacs keys are available: `^b` and `^f` for backward and forward, `^p` and `^n` for previous line and next line, so you can continue even if the cursor keys don't work.

Moving longer distances is done by `\eb` and `\ef` for a word backwards or forwards (or, as you saw, `\M-b` and `\M-f`), and `^a` and `^e` for the start and the end of the line. That just about exhausts the ones you will use the most frequently.

4.2.2 Deleting

For deleting, backspace or the delete key will delete backwards. There is an eternal battle over these keys owing to the fact that on PC keyboards the key at the top left of the central keyboard section is 'backspace' which is the character `^h`, while on traditional UNIX keyboards it is 'delete' which is the character 127, often written as `^?` (which zsh also understands). When you are in the system's own primitive line editing mode, as with sh (unless your sh is really bash), only one of these is 'bound', although it's not really a key binding, it's a translation made by the system's terminal driver, and it's usually the wrong

one. Hence you often find the system prints '^h' on the screen when you want it to delete. You can change the key using

```
stty erase '^h'
```

but zsh protects you from all that — both ^h (backspace) and ^? (delete) will delete backwards one character. Note, by the way, that zsh doesn't understand smart names for any keystrokes – if you try to bind a key called 'backspace' zsh will bind a command to that sequence of characters, not a key of that name. See comments on 'bindkey -s' for when something like this might even be useful.

To confuse matters further, the key often marked as 'Delete' on a 101- or 102-key PC keyboard in the group of 6 above the cursor keys is completely different again, and probably doesn't send either of those sequences. On my keyboard it sends the sequence '\e[3~'. I find it convenient to have this delete the next charater, which is its tradional role in the PC world, which I do by

```
bindkey '\e[3~' delete-char
```

However, the tradtional *Emacs* way of deleting the next character is to use '^d', which zsh binds for you by default. If you look at the binding, which you can do by not giving bindkey an editor command to bind,

```
% bindkey '^d'
delete-char-or-list
```

you'll see it doesn't *quite* do what I suggested. The '-or-list' part is for completion, and you'll find out about it in the next chapter. The first shell I know of to have this odd combination was tcsh.

Since I enjoy confusion, I might as well point out that usually ^d has another use, which is to tell the terminal driver you've reached the end of a file. In the case of, say, a file on a disk, the system knows this by itself, but if you are supplying a stream of characters, the only way of telling it is to send a special character. The default is usually ^d. You will notice that if you type '^d' at the start of the line, you see the message

```
zsh: use 'exit' to exit.
```

That's because zsh recognises the ^d as end-of-file in that position. By default the shell warns you; you can turn this off by setting the option IGNORE_EOF. You can tell the system you don't ever want to send an end-of-file in this way with stty, again: the following are equivalent in Linux but your system way want one or the other:

```
stty eof '^-'
stty eof undef
```

Remember that stty is not part of the shell; it's a way of controlling the state of the system's terminal driver. This means it survives as long as the terminal or terminal window is still connected, even if you start a new shell or exit one that isn't the login shell.

By the way, if you need to refer to a character by its number, the easiest way is probably to use the syntax '\x??', where the '??' are the two hex digits for the key. In the case of delete, it is '\x7f'. You can confirm this by:

```
% bindkey '\x7f'
"^?" backward-delete-char
```

4.2.3 More deletion

You can delete larger areas with '`\ed`' to delete the next word and '`\e^h`' or '`\e^?`' (escape followed by delete backwards) to delete the previous word. '`^u`' usually removes the entire line, before and after the cursor — this is not like Emacs, where `^u` introduces digit arguments as I will describe in the next subsection. It is, however, like another of those primitive editing commands the terminal driver itself provides, this one being known to `stty` as '`kill`'. The most common use of this outside zsh is for deleting your password when you login, when you know you've typed it wrong but can't see how many !@?*! characters you've typed, and maybe can't rely on the terminal agreeing with you as to which of `^h` or `^?` will delete a single one.

Strictly speaking, all the keystrokes in the previous paragraph perform a 'kill' (zsh-speak, not to be confused with the `stty` 'kill') rather than a 'delete' (or deletion, as we used to say when we had a distinct between nouning and verbing). The difference is the same as in Emacs — 'killed' text is saved for later 'yanking' back somewhere else, which you do with the `^y` key, whereas 'deleted' text as with `^?` and `^d` is gone forever. This is what everyone not brought up under Emacs calls 'cut' and 'paste'[1]. Another feature borrowed from Emacs is that if you do multiple 'kills' without any other editing in between, the killed text is joined together and you can yank it all back in one go. I will say more when I talk about point and mark (another Emacs idea).

Actually, even deleted text isn't gone forever: zsh has an Emacs-like editing history, and you can undo the previous commands on the line. This is usually bound to `^xu` and `^x^u`, and there is shorter binding which is described rather confusingly as '`^_`' — confusingly, because on all not-completely-spaced-out keyboards I've ever used you actually generate that sequence by holding down control and pressing the '`/`' key. Zsh doesn't use `^z` by default and, if you are used to Windows, that is another suitable binding for `undo`.

Zsh scores in one way over Emacs — it also has '`redo`', not bound by default. This means that if you undo to much, you can put back what you just undid by repeatedly using the `redo` command.

4.3 Fancier editing

4.3.1 Options controlling zle

Unlike completion, `zle` doesn't have many options associated with it; most of the control is done by key bindings and builtin commands. Only two are really useful; both control beeps. The option `beep` can be unset to tell the shell never to make a noise on an error; the option `histbeep` can be unset to disable beeps only in the case of trying to go back before the first or forward after the last history entry.

The not-very-useful options are `zle` and `singlelinezle`. The former controls whether zle is active at all and isn't that useful because it's usually on automatically whenever you need it, in other words in interative shells, and off whenever you don't. It's sometimes useful to

[1] although since Emacs dates back to the seventies, it could be everyone else that's wrong

test via '`[[ -o zle ]]`', however; this lets you make a function do something cleverer in an interative shell.

The option `singlelinezle` restricts editing to one line; if it gets too long, it will be truncated and a '`$`' printed where the missing bits are. It's only there for compatibility with ksh and as a safeguard if your terminal is really screwed up, though even in that case zsh tries to guess whether everything it needs is available.

Other functions that affect zle include the history functions. These were described back in chapter 2; once you've set it off, searching through the history works basically the same way in zle as with the '`!`' history commands.

4.3.2 The minibuffer and extended commands

The 'minibuffer' is yet another Emacs concept; it is a prompt that appears just under the command line for you to enter some edit required by the editor itself. Usually, it comes and goes as it pleases and you don't need to think about it. The most common uses are entering text for searches, and entering a command which isn't bound to a string. That's yet another Emacs feature: `\ex` prompts you to enter the name of a command. Luckily, since the names tend to be rather long, completion is available. So typing '`echo foo<ESC>xba<TAB>w<TAB>`' ends up with:

```
% echo foo
execute: backward-word
```

and hitting return executes that function, taking you to the start of the `foo`; you might be able to think of easier ways of doing that. This does provide a way of running commands you don't often use.

(I hope my notation isn't too confusing. I write things like `<TAB>` when I'm showing a single character you hit, to make it stand out from the surrounding text. However, when I'm not showing text being entered, I would write that as '`\t`', which is how you would enter the character into a key sequence to be bound, or a string to be printed.)

The minibuffer only handles a very limited set of editing commands. Typing one it doesn't understand usually exits whatever you were trying to do with the minibuffer, then executes the keystroke. However, in this particular case, it won't let you exit until you have finished typing a command; your only other option is to abort. The usual zle abort character is `^g`, '`send-break`'. This is different from the more drastic `^c`, which sends the shell itself an interrupt signal. Quite often they have the same effect in zle, however. (You'll notice `^c` is actually 'bound to `undefined-key`', in other words zle doesn't consider it does anything. However, the terminal driver probably causes it to send an interrupt, and zle does respond to that.)

Another feature useful with rare commands is '`where-is`'. Surprise! it's not bound by default, so typing '`<ESC>xwhere-is`' is then the way of runing it. Then you type another editor command at the '`Where is:`' prompt, and the shell will tell you what keystrokes, if any, are bound to it. You can also simply use `grep` on the output of `bindkey`, which, with no arguments, lists all bindings.

4.3.3 Prefix (digit) arguments

Many commands can be repeated by giving them a numeric prefix or digit argument. For example, at the end of a long line of text, type '<ESC>4<ESC>b'. The '<ESC>b' on its own would take you one word backwards. The '<ESC>4' passes it the number four and it moves four words backwards. Generally speaking, this works any time it make sense to repeat a command. It works for self-insert, too, just repeatedly inserting the character. If it doesn't work, the prefix argument is simply ignored.

You can build up long or negative arguments by repeating both the \e and the digit or '–' after it; for example, '<ESC>-<ESC>1<ESC>0' specifies minus ten. It varies from command to command how useful negative numbers are, but they generally switch from backwards to forwards or similar: '<ESC>-<ESC>4<ESC>\f' is a pointless way of executing the same as '<ESC>4<ESC>b'.

The shell also has Emacs' 'universal-argument' feature, but it's not bound by default — in Emacs it is \C-u, but as we've seen that's already in use. This is an alternative to all those escapes. If you bind the command to a keystroke (it's absolutely pointless as a shortcut otherwise), and type that key, then an option minus followed by any digits are remembered as a prefix. The next keystroke which is not one of those is then executed as a command, with the prefix formed by the number typed after universal-argument.

For example, on my keyboard, the key F12 sends the key sequence '\e[[24~' — see below for how to find out what functions keys send. Hence I use

```
bindkey '\e[[24~' universal-argument
```

Then if I hit the characters F12, 4, 0, a, a row of forty 'a's is inserted onto the command line. I'm not claiming this example is particularly useful.

4.3.4 Words, regions and marks

Words are handled a bit differently in zsh from the way they are in most editors. First, there is a difference between what Emacs mode and vi mode consider words. That is to say, there is a difference between the functions bound by default in those modes; you can use the same functions in either mode by rebinding the keys.

In both vi and Emacs modes, the same logic about words applies whether you are moving forward or backward a number of words, or deleting or killing them; the same amount of text is removed when killing as the cursor would move in the other case.

In vi mode, words are basically the same as what vi considers words to be: a sequence of alphanumeric characters together with underscores — essentially, characters that can occur in identifiers, and in fact that's how zsh internally recognises vi 'word characters'. There is one slight oddity about vi's wordwise behaviour, however, which you can easily see if you type '/a/filename/path/', leave insert mode with ESC, and use 'w' or 'b' to go forward or backward by words over it. It alternates between moving over the characters in a word, and the characters in the separator '/'.

In Emacs, however, it is done a bit differently. The vi 'word characters' are always considered parts of a word, but there is a parameter $WORDCHARS which gives a string of characters which are *also* part of a word. This is perhaps opposite to what you would expect; given that alphanumerics are always part of a word, you might expect there to be a parameter to which you add characters you *don't* want to be part of a word. But it's not like that.

Also unlike vi, jumping a word always means jumping to a word character at the start of a word. There is no extra 'turn' used up in jumping over the non-word characters.

The default value for `$WORDCHARS` is

```
*?_-.[]~=/&;!#$%^(){}<>
```

i.e. pretty much everything and the kitchen sink. Usually, therefore, you will want to remove characters which you don't want to be considered parts of words; '–', '/' and '.' are particularly likely possibilities. If you want to remove individual characters, you can do it with some pattern matching trickery (next chapter):

```
% WORDCHARS=${WORDCHARS//[&.;]}
% print $WORDCHARS
*?_-[]~=/!#$%^(){}<>
```

shows that the operation has removed those three characters in the group, i.e. '`&`', '`.`' and '`;`', from `$WORDCHARS`. The '`//`' indicates a global substitution: any of the characters in the square brackets is replaced by nothing.

Many other line editors, even those like `readline` with Emacs bindings, behave as if only identifier characters were part of a word, i.e. as if `$WORDCHARS` was empty. This is very easy to do with a zle shell function. Recent versions of zsh supply the functions '`bash-forward-word`', '`bash-kill-word`', and a set of other similar ones, for you to bind to keys in order to have that behaviour.

Other behaviours are also possible by writing functions; for example, you can jump over real shell words (i.e. individual command arguments) by using some more substitution trickery, or you can consider only space-delimited words (though that's not so far from what you get with `$WORDCHARS` by adding '`" ' \@`').

4.3.5 Regions and marks

Another useful concept from Emacs is that of regions and marks. In Emacs-speak 'point' is where the cursor is and 'mark' is somewhere where you leave a mark to come back to later. The command to set the mark at the current point is '`^@`' as in Emacs, a hieroglyphic which usually means holding down the control key and pressing the space key. On some systems, such as the limited version of `telnet` provided with a well-known non-UNIX-based windowing system, you can't send this sequence, and you need to bind a different sequence to `set-mark-command`. One possibility is '`\e `' (escape followed by space), as in MicroEMACS. (Some X Windows configurations don't allow `^@` to work in an xterm, either, though that is usually fixable.)

To continue with Emacs language, the region between point and mark is described simply as 'the region'. In zsh, you can't have this highlighted, as you might be used to with editors running directly under windowing systems, so the easiest way to find out the ends of the region is with `^x^x`, `exchange-point-and-mark`, which I mentioned before — mark, by default, is left at the beginning of the line, hence the behaviour you saw above.

Various editing commands — usually those with '`region`' in the name — operate on this. The most usual are those which kill or copy the region. Annoyingly, `kill-region` isn't bound — in Emacs, it's `^w`, but zsh follows the tradition of having that bound to

backward-kill-word, even though that's also available as the traditional Emacs binding \e^?. So it's probably useful to rebind it. To copy the region, the usual binding '\ew' works.

You then 'yank' back the text copied or killed at another point with '^y'. The shell implements the 'kill ring' feature, which means if you perform a yank, then type '<ESC>y' (yank-pop) repeatedly, the shell cycles back through previously killed or copied text, so that you have more available than just the last one.

4.4 History and searching

Zle has access to the lines saved in the shell's history, as described in 'Setting up history' in chapter 2. There are essentially three ways of retrieving bits of the history: moving back through it line by line, searching back for a matching line, and extracting individual words from the history. In fact, the first two are pretty similar, and there are hybrid commands which allow you to move back step by step but still matching only particular lines.

4.4.1 Moving through the history

The simplest behaviour is what you get with the normal cursor key bindings, 'up-line-or-history' and 'down-line-or-history'. If you are in text which fits into a single line (which may be a continuation line, i.e. it has a new prompt in the form given by $PS2 at the start of the line), this replaces the entire line with the line before or after in the history. The history is not circular, it has a beginning and an end. The beginning is the first line still remembered by the shell (i.e. $HISTSIZE lines back, taking into account that the actual number of lines present will be modified by the effects of any special history options you have set to remove unwanted lines); the end is the line you are typing. You can use \e< and \e> to go to the first and last line in the history.

The last phrase sounds trivial but isn't quite. Type 'echo This is the last line', go back a few lines with the up arrow, and then back down to the end, and you will see what I mean — the shell has remembered the line you were typing, even though it hadn't been entered, so you can scroll up and down the history and still come back to it.

Of course, you can edit any of the earlier history lines, and hit 'return' so that they are executed — that's the whole point of being able to scroll back through the history. What is maybe not so obvious is that the shell will remember changes you make to these lines, too, until you hit 'return'.

For example, type 'echo this is the last line' at a new shell prompt, but don't hit return. Now hit the up arrow once, and edit the previous line to say 'echo this is the previous line'. If you scroll down and up, you will see that the shell has kept both of those lines. When you decide which one to use and hit return, that line is executed and added to the end of the history, and any changes to previous lines in the history are forgotten.

Sometimes you don't want to add a new line to history, instead re-execute a whole series of earlier commands one by one. This can be done with ^o, accept-line-and-down-history. When you hit ^o on a line in the history, that is executed, and the line after it in the history is shown. So you just need to keep hitting it to keep executing the commands.

There are two other similar commands I don't use as much, infer-next-history, bound to ^x^n, and accept-and-infer-next-history, not bound by default. 'Inferring' the next

history means that the shell looks at what is in the current line, whatever its provenance — you might just have typed it, for example — and looks back in the history for a matching line; the 'inferred' next history line is the one following that line. In the first case, you are simply shown that line; in the second case, the current line is executed first, then you are shown the inferred line. Feel free to drop me a line if you find this is the best thing since sliced bread.

One slight confusion about the history is that it can be hard to remember quite where you are in it, for example, if you were editing a line and had to scroll back to look for something else. In cases like this, `\e>` is your friend, as it takes you the last line. Also, whenever you hit return, you are guaranteed to be at the end of the history, even if you were editing a line some back in the history, unlike certain other systems (though `accept-line-and-down-history` can emulate those). So it's usually not too hard to stay unconfused about what you're editing.

4.4.2 Searching through the history

Zsh has the commands you would expect to search through the history, i.e. where you hit a search key and then type in the words to search for. However, it also has other features, probably more used by the zsh community, where the search is based on some feature of the current line, in particular the first word or the line up to the cursor position. These typically enable you to search backwards more quickly, since you don't need to tell the shell what you are looking for.

Ordinary searching

The standard search commands, by which I mean the ones your are probably most familiar with from ordinary text editors (if either Emacs or vi can be so called), are designed to make Emacs and vi users feel at home.

In Emacs mode, you have incremental search: `^r` to search backwards — this is usually what you want, since you usually start at the end — and `^s` to search forwards. Note that `^s` is another keystroke which is often intercepted by the terminal driver; in this case, it usually freezes output to the terminal until you type `^q` to turn it back on. If you don't like this, you can either use '`stty stop`' and '`stty start`' to change the characters, or simply '`unsetopt flowcontrol`' to turn that feature off altogether. However, the command bound to `^s`, `history-incremental-search-forward`, is also bound to `^xs`, so you can use that instead.

As in Emacs, for each character that you type, incremental search takes you to the nearest history entry that matches all the characters, until the match fails. Typing the search keystroke again at any point takes you to the next match for the characters in the minibuffer.

In vi command mode, the keystrokes available by default are the familiar '`/`' and '`?`'. There are various differences from vi, however. First of all, it is '`/`' that searches backwards — this is the one you will use more often. Secondly, you can't search for regular expressions (patterns); the only exception is that the character '`^`' at the start anchors the search to the start of a line. Everything else is just a plain string.

The other two standard vi search keystrokes are also present: '`n`' searches for the next match of the current string, and '`N`' does the same but reverses the direction of the search.

Search for the first word

The next sort of search is probably the most commonly used, but is only bound in Emacs mode: `\ep` and `\en` search forward or backward for the next history line with the same first word as the current line. So often to reuse a command you will type just the command name itself, and hit `\ep` until the command line you want appears. These commands are called simply '`history-search-backward`' and '`history-search-forward`'; the name doesn't really describe the function all that well.

Prefix searching

Finally, you can search backwards for a line which has the entire starting point up to the cursor position the same as the current line. This gives you a little more control than `history-search-`*direction*. The corresponding commands, `history-beginning-search-backward` and `history-beginning-search-forward`, are not bound by default. I find it useful to have them bound to `^xp` and `^xn` since this is similar to the initial-word searches:

```
bindkey '^xp' history-beginning-search-backward
bindkey '^xn' history-beginning-search-forward
```

Other search commands based on functions

Search commands are one of the types most often customised by writing shell functions. Some are supplied with the latest versions of the shell; have a look in the ZLE section of the `zshcontrib` manual page. You should find the functions themselves installed somewhere in your `$fpath`, typically

```
/usr/local/share/zsh/$ZSH_VERSION/functions
```

or in the subdirectory `Zle` of that directory, depending how your version of zsh was installed. If the shell was pre-installed, the most likely location is

```
/usr/share/zsh/$ZSH_VERSION/functions/Zle
```

These should guide you in writing your own.

One point to note is that when called from a function the `history-search-`*direction* and `history-incremental-search-`*direction* can take a string argument specifying what to search for. In the first case, this is just a one off search, while in the second, you remain in incremental search and the string is used to prime the minibuffer, so you can edit it. I will later say much more about writing zle functions, but calling a search command from a user-defined editing function is as simple as:

```
zle history-search-backward search-string
```

and you can test the return status to see if the search succeeded.

4.4.3 Extracting words from the history

Sometimes instead of editing a previous line you just want to extract a word from it into the current line. This is particularly easy if the word is the last on the line, and the line isn't far back in the history: just hit `\e.` repeatedly, and the shell will cycle through the last word on previous lines. You can give this a prefix argument to pick the *N*th from last word on the line just above the last line you picked a word from. As you can tell from the description, this gets a little hairy; version 4.1 of the shell will probably provide a slightly more flexible version.

Although it's strictly not to do with the history, you can copy the previous word on the current line with `copy-prev-word`, which for some reason is bound to `\e^_`, escape followed (probably) by control and slash. I have this bound to `\e=` instead (in some versions of ksh that key sequence is taken by the equivalent of `list-choices`). This copies words delimited by whitespace, but you can copy what the shell would see as the previous complete argument by using `copy-prev-shell-word` instead. This isn't bound by default, as it is newer than the other one, but it is arguably more useful.

Sometimes you want to complete a word from the history; this is possible using the completion system, described in the next chapter.

4.5 Binding keys and handling keymaps

There are two topics to cover under the heading of key bindings: first, how to bind keys themselves, and secondly, keymaps and how to use them. Manipulating both key bindings and keymaps is done with the `bindkey` command. The first topic is the more immediately useful, so I'll start with that.

4.5.1 Simple key bindings

You've already seen basic use of `bindkey` to link editing commands to a particular sequence of keys. You've seen the shorthand for naming keys, with `\e` being escape and `^x` the character `x` pressed while the control key is held down. I even said something about 'meta' key bindings.

Let me now launch into a little more detail. When you bind a key sequence, which you do with '`bindkey` *key-sequence editor-command*', the *key-sequence* can consist of as many characters as you like. It doesn't even matter (much) if some initial set of the key sequence is already bound. For example, you can do,

```
bindkey '\eA' backward-word
bindkey '\eAA' beginning-of-line
```

Here, I'll follow the shell documentation in referring to `\eA` as the prefix of `\eAA`.

This introduces two points. First, note that the binding for `\eA` is distinct from that for `\ea`; you will see the latter still does `accept-and-hold` (in Emacs mode), which means it excutes the current line, then gives it back to you for editing — useful for doing a lot of quite similar tasks. Meanwhile, `\eA` takes you back a word.

This case sensitivity only applies to alphabetic characters which are a complete key in their own right, not to those characters with the control key held down — `^x` and `^X` are identical. (You may have found there are ways to bind both separately in Emacs when running under a windowing system, since the windowing system can tell Emacs if the shift key is held down with the others; it's not that simple if you are using an ordinary terminal.)

If you entered both those `bindkey` commands, you may notice that there is a short pause before `\eA` takes effect. That's because it's waiting to see if you type another `A`. If you do type the extra `A` during that pause, you will be taken to the beginning of the line instead. That pause is how the shell decides whether to execute the prefix on its own.

The time it waits is configurable and is given by the parameter `$KEYTIMEOUT`, which is the delay in hundredths of a second. The default is 40, i.e. four tenths of a second. Its use is usually down to personal preference; if you don't type very fast, you might want to increase it, at the expense of a longer delay when you are waiting for the prefix to be executed. If you are editing on a remote machine over a very slow link, you also may need to increase it to be able to get full key sequences which have such a prefix to work at all.

However, the shell only has this ambivalent behaviour if a prefix is bound in its own right; if the initial key or keys don't mean anything on their own, it will wait as long as you like for you to type a full sequence which is bound. This is by far the normal case. The only common example of a separately bound prefix is in vi insert mode, where `<ESC>` takes you back to command mode, while there may be other bindings starting with `\e` such as the cursor keys. We'll see below how you can remove those if they offend your sense of vi purity. (Don't laugh, vi users are strange.)

Note that if the whole sequence is not bound, after all, the shell will abort as soon as it reads a complete key sequence which is no longer a prefix. For example, if you type '`\e[`', the chances are the shell is waiting for more, but if you add a '`/`', say, it will probably decide you are being silly and abort. The next key you type then starts a new sequence.

4.5.2 Removing key bindings

If you want to remove a key binding, you can simply bind it to something else. Near all uses of the `bindkey` and `zle` commands are smart about removing dead wood in such cases. However you can also use '`bindkey -r` *key-sequence*' to remove the binding explicitly. You can also simply bind the sequence to the command `undefined-key`; this has exactly the same effect — even down to pruning completely any bindings for long sequences. For example, suppose you bind '`\e\C-x\C-x`' to a command, then to `undefined-key`. All memory that '`\e\C-x\C-x`' was ever bound is removed; `\e\C-x` will no longer be marked as a prefix key, unless you had some other binding with that prefix.

You can remove all bindings starting with a given prefix by adding the '`-p` option. The example given in the manual,

```
bindkey -rpM viins '\e'
```

(except it uses the equivalent form '`^[`') is amongst the most useful, as it will remove the annoying delay after you type '`\e`' to enter vi command mode. The delay is there because the cursor keys usually also start with `\e` and the shell is waiting to see if you actually typed one of those. So if you can make do without cursor keys in vi insert mode you may want to consider this.

Note that any binding for the prefix itself is not removed. In this example, `\e` stays bound

as it was in the `viins` keymap, presumably to `vi-cmd-mode`.

All manipulations like this are specific to one particular keymap. You need to repeat them with a different `-M ...` option argument, which is described below, to have the same effect in other keymaps.

4.5.3 Function keys and so on

It's usually possible to bind the function keys on your keyboard, including the specially named ones such as 'Home' and 'Page Up'. It depends a good deal on how your windowing system or terminal driver handles them, but these days it's nearly always the case that a well set-up system will allow the function keys to send a string of characters to the terminal. To bind the keys you need to find out what that string is.

Luckily, you are usually aided by the fact that only the first character of the string is 'funny', i.e. does something other than insert a character. So there is a trick for finding out what the sequence is. In a shell window, hit `^v` (if you are using vi bindings, you will need to be in insert mode), then the function key in question. You will probably see a string like '`^[OP`' — this is what I get from the F1 key. A note in my `.zshrc` suggests I used to get '`\e[11~`', so be prepared for something different, even if, like me, you are using a standard xterm terminal emulator. A quick poll of terminal emulators on this Linux/GNU/XFree86 system suggests these two possibilities are by far the most popular.

You may even be able to get different sequences by holding down shift or control as well (after pressing `^v`, of course). On my keyboard, combining F1 with shift gives me '`^[O2P`', with control '`^[O5P`' and with both '`^[O6P`'. Again, your system may do something completely different.

If you move the cursor back over that '`^[`', you'll find it's a single character — you can position the cursor over the '`^`', but not the '`[`'. This is zsh's way of inserting a real, live escape character into the line. In fact, if you type

```
bindkey '
```

then `^v`, the function key, and the other single quote, you have a perfectly acceptable way of binding the key on the command line. Zsh is generally quite relaxed about your use of unprintable characters; they may not show up correctly on your terminal, but the shell is able to handle all single-byte characters. It doesn't yet have support for those longer than a single byte, however.

You can also do the same in your `.zshrc`; the shell will handle strange characters in input without a murmur. You can also use the two characters '`^[`', which is just another way of entering an escape key. However, the kosher thing to do is to turn it into '`\e`'. For example,

```
bindkey '\e[OP'  where-is           # F1
bindkey '\e[O2P' universal-argument # shift F1
```

and so on. Using this, you can give sensible meanings to 'Home', 'End', etc. Note the windowing system's sensible way of avoiding the problem with prefixes — any extra characters are inserted before the final character, so the shell can easily tell when the sequence is complete without having to wait and see if there is more to follow.

There is a utility supplied with zsh called `zkbd` which can help with all of this by finding out and remembering the definitions for you. You can probably use it simply by autoloading it

and running it, as it is usually installed with the other functions. It should be reasonably self-explanatory, else consult the `zshcontrib` manual.

If you are using X Windows and are educated enough, you can tinker with your `.Xdefaults` file to tweak how keys are interpreted by the terminal emulator. For example, the following turns the backspace key into a delete key in anything using a 'VT100 widget', which is the basis of xterm's usual mode of operation:

```
*VT100.Translations: #override \
<Key>BackSpace: string(0x7F)
```

Part of the reason for showing this is that it makes zsh's key binding system look wonderfully streamlined by comparison. However, tinkering around at this level gives you very much more control over the use of key modifiers (shift, alt, meta, control, and maybe even super and hyper if you're lucky). This is far beyond the scope of this guide — which I say, as you probably realise by now, to cover up for not knowing much about it. Here's another example from Oliver Kiddle, though; it uses control with the left-cursor key to send an escape sequence: insert

```
Ctrl<Key>Left: string(0x1b) string("[159q") \n\
```

into the middle of the example above — this shows how multiple definitions are handled. Modern xterms already send special escape sequences which you can investigate and bind to as I've described.

4.5.4 Binding strings instead of commands

It's possible to assign an arbitrary string of characters to a key sequence instead of an editor command by giving `bindkey` the option `-s`. One of the good things about this is that the string of characters are reinterpreted by zle, so they can contain active key sequences. In the old days, this was quite often used as a basic form of macro, to string together editor commands. For example, the following is a simple way of moving backward two words by repeating the Emacs mode bindings. I've used my F1 binding again; yours may be completely different.

```
bindkey -s '\e[OP' '\eb\eb'
```

It's not a good idea to bind a key sequence to another string which includes itself.

This method has the obvious drawback that if someone comes along and rebinds '`\eb`', then F1 will stop working, too. Nowadays, this sort of task can be done much more flexibly and clearly by writing a user-defined widget, which is described in a later section. So bindings of this sort are going a little out of fashion. However, they do provide quick shortcuts. Two from Oliver Kiddle:

```
bindkey -s '^[[072q' '^V^I'                          # Ctrl-Tab
bindkey -s "\C-x\C-z" "\eqsuspend\n"
```

You can also quite easily do some of the things you can do with global aliases.

Remember that 'ordinary' characters can be rebound, too; they just usually happen to have a binding which makes them be inserted directly. As a particularly pointless example, consider:

```
bindkey -s secret 'Oh no!'
```

If you type '`secret`' fast enough the letters are swallowed up and '`Oh no!`' appears instead. If you pause long enough anywhere in the middle, the word is inserted just as normal. That's because all parts of it can be interpreted as prefixes in their own right, so `$KEYTIMEOUT` applies at every intervening stage. Less pointlessly, you could use this as a way of defining abbreviations.

4.5.5 Keymaps

So far, all I've said about keymaps is that there are three standard ones, one for Emacs mode and two for vi mode, and that '`bindkey -e`' and '`bindkey -v`' pick Emacs or vi insert mode bindings. There's no simple way of picking vi command mode bindings, since that's not usually directly available but entered by the `vi-cmd-mode` command, usually bound to `\e`, in vi insert mode. (There is a '`bindkey -a`', but that doesn't pick the keymap for normal use; it's equivalent to, but less clear than, '`bindkey -M vicmd`'.)

Most handling of keymaps is done through `bindkey`. The keymaps have short names, `emacs`, `viins` and `vicmd`, for use with `bindkey`. There is also a keymap `.safe` which you don't usually need but which never changes, so can be used if your experimentation has completely ruined every other keymap. It only has bindings for `self-insert` (most keys) and `accept-line` (`^j` and `^m`), but that's enough to enter commands.

The names are most useful in two places. First, you can use '`bindkey -M` *keymap*' to define keys in a particular map:

```
bindkey -M vicmd "\e[OA" up-line-or-history
```

binds the usual up-cursor key in `vicmd` mode, whatever keymap is currently set. Actually, any version of the shell which understands the `-M` option probably has that bound already.

Secondly, you can force zle to use a particular keymap. This is done in a slightly non-obvious way: zle always uses the keymap `main` as the current keymap (except when it's off in vi command mode, which is handled a bit specially). To use your own, you need to make `main` an alias for that with '`bindkey -A`'. The order after this is the same as that after `ln`: the existing keymap you want to refer to comes first, then what you want to make an alias for it, in this case `main`. This means that

```
bindkey -A emacs main
```

has the same effect as

```
bindkey -e
```

but is more explicit, if a little more baroque. Don't link `vicmd` to main, since then you can't use `viins`, which is bad. Note that '`bindkey -M emacs`' doesn't have this effect; it simply lists the bindings in the `emacs` keymap.

You can create your own keymaps, too. The easiest way is to copy an existing keymap, such as

```
bindkey -N mymap emacs
```

which creates (or replaces) `mymap` and initialises it with the bindings from `emacs`. Now you can use `mymap` just like `emacs`. The bindings in each are completely separate. If you finish with a keymap, you can remove it with '`bindkey -D keymap`', although you'd better make sure it's not linked to `main` first.

You can omit '`emacs`' to create an empty keymap; this might be appropriate if your keymap is only going to be used in certain special places and you want complete control on what goes into it. Currently the shell isn't very good at letting you apply your own keymaps just in certain places, however.

There are various other keymaps you might encounter, used in special circumstances. If you list all keymaps, which is done by '`bindkey -l`', you may see `listscroll` and `menuselect`. These are used by the new completion system, so if that isn't active, you probably won't see them. They reside in the module `zsh/complist`. There will be more about their effects in chapter 6; `listscroll` allows you to move up and down completion lists which more than fill the terminal window, and `menuselect` allows you to select items interactively from a displayed list. You can bind keys in them as with any other keymap.

4.6 Advanced editing

(In physics, the 'advanced wave' is a hypothetical wave which moves backwards in time. Unfortunately, however useful it would be to meet deadlines, that's not what I meant by 'advanced editing'.)

Here are are a few bits and pieces which go beyond ordinary line editing of shell commands. Although they haven't been widespread in shells up to now, I use all of them every day, so they aren't just for the postgraduate zsh scholar.

4.6.1 Multi-line editing

All Bourne-like shells allow you to edit continuation lines; that is, if the shell can work out for sure that you haven't finished typing, it will show you a new prompt, given by `$PS2`, and allow you to continue where you left off from the previous line. In zsh, you can even see what it is the shell is waiting for. For a simple example, type '`array=(first`' and then 'return'. The shell is waiting for the final parenthesis of the array, and prints '`array>`' at you, unless you have altered `$PS2`. You can continue to add elements to the array until you close the parentheses.

Shells derived from csh are less happy about continuation lines; historically, this is because they try to evaluate everything in one go, and became confused if they couldn't. The original csh didn't have a particularly sophisticated parser. For once, zsh doesn't have an option to match the csh behaviour; you just have to get used to the idea that things work in zsh.

Where zsh improves over other shells is that you aren't just limited to editing a single continuation line; you can actually edit a whole block of lines on screen as you would in

a full screen editor — although you can't scroll off the chunk of lines you're editing, which wouldn't make sense.

The easiest way of doing this is to hit escape before you type a newline at the point where you haven't finished typing. Actually, you can do this any time, even if the line so far is complete. For example,

```
% print This is line one<ESC><RET>
print This is line two
```

where those angle brackets at the end of the line means you type escape, then return. Nothing happens, and there is no new prompt; you just type blithely on. Hit return, unescaped this time, and both lines will be executed. Note there is no implicit backslash, or anything like that; when zsh reads the whole caboodle, that escaped carriage return became a real carriage return, just as the shell would have read it from a script.

This works because '`\e\r`' is actually bound to the command `self-insert-unmeta`' which means 'insert the character you get by stripping the escape or top bit from what I just typed' — in other words, a literal carriage return. You would have got exactly the same effect by typing `^v^j`, since the `^v` likewise escapes the `^j` (newline), as it does any other character.

(Aside for the terminally curious only: Why newline here and not carriage return — the 'enter' key — as you might expect? That's a rather grotesque story. It turns out that for mostly historical reasons UNIX terminal drivers like to swap newline and carriage return, so when you type carriage return (sent both by that key and by `^m`, which is the same as the character represented by `\r`), it comes out as newline (on most keyboards, just sent by `^j`, which is the same as the character represented by `\n`). It is the newline character which is the one you 'see' at the end of the line (by virtue of the fact it is the end of the line). However, `^v` sees through this and if you type `^m` after it, it inserts a literal `^m`, which just looks like a `^m` because that's how zsh outputs it. So that's why that doesn't work. Actually, `self-insert-unmeta` would see the `^m`, too, because that's what you get when you strip off the `\e`, but it has a little extra code to make UNIX users feel at home, and behaves as if it were a newline. Normally, `^j` and `^m` are treated the same way (`accept-line`), but the literal characters have different behaviours. If you're now very confused, just be thankful I haven't told you about the additional contortions which go on when outputting a newline.)

It probably doesn't seem particularly useful yet, because all you've done is miss out a new prompt. What makes it so is that you can now go up and down between the two (or more) lines just using the cursor keys. I'm assuming you haven't rebound the cursor keys, your terminal isn't a dumb one which doesn't support cursor up, and the option `singlelinezle` isn't in effect — just unset it if it is, you'll be grateful later.

So for example, type

```
% if [[ true = false ]]; then<ESC><RET>
  print Fuzzy logic rules<ESC><RET>
fi
```

where I indented that second line just with spaces, because I usually do inside an 'if'. There are no continuation prompts here, just the original `$PS1`; that's not a misprint. Now, before hitting return, move up two lines, and edit `false` to `true`. You can see how this can be useful. Entering functions at the command line is probably a more typical example.

Suppose you've already gone through a few continuation lines in the normal way with `$PS2`'s? You can't scroll back then, even though the block hasn't yet been edited. There's

a magic way of turning all those continuation lines into a single block: the editor command `push-line-or-edit`. If you're not on a continuation line, it acts like the normal `push-line` command, which we'll meet below, but for present purpose you use it when you are on a continuation line. You are presented with a seamless block of text from the (redrawn) prompt to the end which you can edit as one. It's quite reasonable to bind `push-line-or-edit` instead of `push-line`, to either `^q` or `\eq` (in Emacs mode, which I will assume, as usual). Be careful with `^q`, though — if the option `flowcontrol` is set it will probably be swallowed up by the terminal driver and not get through to the shell, the same problem I mentioned above for `^s`.

4.6.2 The builtin vared and the function zed

I mentioned the `vared` command in chapter 3; it uses the normal line editor to editor a variable, typically a long one you don't want to have to type in completely like `$path`, although you need to remember *not* to put the '`$`' in front or the shell will substitute it before `vared` is run. However, since it's just a piece of text like any other input, this, too, can have multiple lines, which you enter in the same way — and since a shell parameter can contain anything at all, you have a pretty general purpose editor. The shell function '`zed`' is supplied with the shell and allows you to edit a file using all the now-familiar commands. Since when editing files you don't expect a carriage return to dump you out of the editor, just to insert a new line, zed rebinds carriage return to `self-insert-unmeta` (the '`-unmeta`' here is just to get the swapping behaviour of turning the carriage return into a newline). To save and exit, you can type `^j`, or, if your terminal does something odd with that, you can also use `^x^w`, which is designed to look like Emacs' way of writing a file.

If you look at `zed`, you will see it has a few bells and whistles — for example, '`zed -f`' allows you to edit a function — but the code to read a file into a parameter, edit the parameter, and write the parameter back to the file is extremely simple; all the hard editing code is already handled within `vared`. Indeed, `zed` is essentially a completely general purpose editor, though it quickly becomes inefficient with long files, particularly if they are larger than a single screen; as you would expect, zle was written to cope efficiently with short chunks of text.

It would probably be nice if you could make key bindings that only applied within vared by using a special keymap. That may happen one day.

By the way, note that you can edit arrays with vared and it will handle the different elements sensibly. As usual, whitespace separates elements; when it presents you with an array which contains whitespace within elements, vared will precede it with a backslash to show it isn't a separator. You can insert quoted spaces with backslashes yourself. Only whitespace characters need this quoting, and only backslashes work.

For example,

```
array=('one word' 'two or more words')
vared array
```

presents you with '`one\ word two\ or\ more\ words`'. If you add '` and\ some\ more.`', hit return, and type '`print -l $array`' to show one element per line you will see

```
one word
two or more words
and some more.
```

Some older versions of the shell were less careful about spaces within elements.

4.6.3 The buffer stack

The mysterious other use for `push-line-or-edit` will now be explained. Let's stick to `push-line`, in fact, since I've already dealt with the `-or-edit` bit.

Type

```
print I was just in the directory
```

(no newline). Oh dear, which directory were you just in? You don't want to interrupt the flow of text to find out. Hit '`\eq`'; the line you've been typing disappears — but don't worry, it hasn't gone. Now type

```
dirs
```

Two things happen: that last line is executed, of course, showing the list of directories on the directory stack (your use of `pushd` and `popd`), but also the line you got rid of before has reappeared, so you can continue to edit it.

You may not realise straight away quite how useful this is, but I used it several times just while I was writing the previous paragraph. For example, I was alternating directories between the zle source code and the directory where I keep this guide, and I started typing a '`grep`' command before realising I was in the wrong directory. All I need to do is type `\eq`, then `pushd`, to put me where I want to be, and finish off the `grep`.

The 'buffer stack', which is the jargon for this mechanism, can go as deep as you like. It's a last-in-first-out (LIFO) stack, so the line pushed onto it by the most recently typed `\eq` will reappear first, followed by the back numbers in reverse order. You can even prime the buffer stack from a function — not necessarily a zle function, though that works too — with '`print -z` *command-line*'.

You can pull something explicitly off the stack, if you want, by typing `\eg`, but that has the same effect as clearing the current line and hitting return. You can of course push the same line multiple times: if you need to do a whole series of things before executing it, just hit `\eq` again each time the line pops back up.

I lied a little bit, to avoid confusion. The cleverness of `push-line-or-edit` about multi-line buffers extends to the this case, too. If you do a normal `push-line` on a multi-line buffer, only the current single line is pushed; the command to push the whole lot, which is probably what you want, is `push-input`. But if you have `push-line-or-edit` bound, you can forget the distinction, since it will do that for you. If you've been paying attention you can work out the following sequence (assuming `\eq` has been rebound to `push-line-or-edit`):

```
% if [[ no = yes ]]; then
then> print<ESC>q<ESC>q
```

The first `\eq` turns the two lines into a single buffer, then the second pushes the whole lot onto the buffer stack. This saves a lot of thinking about bindings. Hence I would recommend users of Emacs mode add

```
bindkey '\eq' push-line-or-edit
```

to their `.zshrc` and forget the distinctions.

4.7 Extending zle

We now come to the newest and most flexible part of zle, the ability to create new editing commands, as complicated as you like, using shell functions. This was originally introduced by Andrew Main ('Zefram') in zsh 3.1 and so is standard in all versions of zsh 4, although work goes on.

4.7.1 Widgets

If you don't speak English as you first language, first of all, congratulations for getting this far. Secondly, you may think of 'widget' only as a technical word applied to the object which realises some computational idea, like the thing that implements text editing in a window system, for example. However, to most English speakers, 'widget' is a humorous word for an object, a bit like 'whatyoumacallit' or 'thingummybob', as in 'where's that clever widget that undoes the foil and takes out the cork in one go'. Zsh's use has always seemed to me closer to the second, non-technical version, but I may be biased by the fact that the internal object introduced by Zefram to represent a widget, and never seen by the user, is called a 'thingy', which I won't refer to again since you don't need to know.

Anyway, a 'widget' is essentially what I've been calling an editor command up to now, something you can bind to a key sequence. The reason the more precise terminology is useful is that as soon as you have shell functions flying around, the word 'command' is hopelessly non-specific, since functions are full of commands which may or may not be widgets. So I make no apology for using the word.

So now we are introducing a second type of widget: one which, instead of something handled by code built into the shell, is handled by a function written by the user. They are completely equivalent; `bindkey` and company don't care which it is. All you need to do to create a widget is

```
zle -N widget-name function-name
```

then *widget-name* can be used in `bindkey`, or `execute-named-cmd`, and the function *function-name* will be run. If the `widget-name` and `function-name` are the same, which is often the simplest thing to do, you just need one of them.

You can list the existing widgets by using '`zle -l`', although often '`zle -lL`' is a better choice since the output format is then the same as the form you would use to define the widget. If you see lots of '`zle -C`' widgets when you do that, ignore them for now; they are completion widgets, handled a bit differently and described in chapter 6.

Now you need to know what should go into the function.

4.7.2 Executing other widgets

The simplest thing you can do inside a function implementing a widget is call an existing function. So,

```
my-widget() {
  zle backward-word
}
zle -N my-widget
```

creates a widget called `my-widget` which behaves in every respect (except speed) like the builtin widget `backward-word`. You can even give it a prefix argument, which is passed down; `\e3` then whatever you bound the widget to (or `\exmy-widget`) will go backward three words.

Suppose you wanted to pass your own prefix argument to `backward-word`, instead of what the user typed? Or suppose you want to take account of the prefix argument, but do something different with it? Both are possible.

Let's take the first of those. You can supply a prefix argument for this command alone by putting `-n` *argument* after the widget name (note this is not where most options go).

```
my-widget() {
  zle backward-word -n 2
}
```

This always goes backwards two words, overriding any numeric argument given by the user. (You can redefine the function without telling zle about it, by the way; zle just calls whatever function happens to be defined when the widget is run.) If you put just `-N` after the name instead, it will cancel out any prefix given by the user, without introducing a new one.

The other part of prefix handling — intercepting the one the user specified and maybe modifying it — introduces one of the most important parts of user-defined widgets. Zle provides various parameters which can be read and often written to alter the behaviour of the editor or even the text being edited. In this case, the parameter is `$PREFIX`. For example,

```
my-widget() {
  zle backward-word -n $(( ${NUMERIC:-1} * 2 ))
}
```

This uses an arithmetic substitution to provide an argument to `backward-word` which is twice what the user gave. Note that `${NUMERIC:-1}` notation, which is important: most of the time, you don't give a numeric argument to a command at all, and in that case zle naturally enough treats `$NUMERIC` as if it wasn't set. This would mess up the arithmetic substitution.

By the way, if you do make an error in a shell function, you won't see it; you'll just get a beep, unless you've turned that off with `setopt nobeep`. The output from such functions is junked, since it would mess up the display. So you should do any basic debugging before turning the function into a widget, for example, stick a `print` in front and run it directly — you can't execute widgets from outside the editor.

The following also works:

```
my-widget() {
  (( NUMERIC = ${NUMERIC:-1} * 2 ))
  zle backward-word
}
```

because you can alter `$NUMERIC` directly, and unless overridden by the `-n` argument it is used by any widgets called from the function. If you called more widgets inside the function — and you can call as many as you like — the same argument would apply to all the ones that didn't have an explicit `-n` or `-N`.

Some widgets allow you to specify non-numeric arguments. At the moment these are mainly search functions, which you can give an explicit search string. Usually, however, you want to specify a new search string each time. The most useful way of using this I can see is to provide an initial argument for incremental search commands. Later, I'll show you how you can read in characters in a similar fashion to Emacs mode's `^r` binding, `history-incremental-search-backwards`.

4.7.3 Some special builtin widgets and their uses

There are some things you might want to do with the editor in a zle function which wouldn't be useful executed directly from zle. One is to cause an error in the same way as a normal widget does. You can do that with '`zle beep`'. However, this doesn't automatically stop your function at that point; it's up to you to return from it.

It's possible to redefine a builtin widget just by declaring it with '`zle -N`' and defining the corresponding function. From now on, all existing bindings which refer to that widget will cause yours to be run instead of the builtin one. This happens because zle doesn't actually care what a widget does until it is run. You can see this by using `bindkey` to define a key sequence to call an undefined widget such as `any-old-string`. The shell doesn't complain until you actually hit the key sequence.

Sometimes, however, you want to be sure to call the builtin widget, even if the behaviour has been redefined. You can do this by putting a '`.`' in front of the name of the widget; '`zle .up-line-or-history`' always calls the builtin widget usually referred to as `up-line-or-history`, even if the latter has been redefined. One use for this is to rebind '`accept-line`' to do something whenever zle is about to pass a line up to the shell, but to accept the line anyway: you write your own widget `accept-line`, make sure it calls '`zle .accept-line` just before it finishes, and then use '`zle -N accept-line`. Here's a trivial but not entirely stupid example:

```
accept-line() {
  print -n "\e]2;Executing $BUFFER\a"
  zle .accept-line
}
zle -N accept-line
```

Now every time you hit return to execute a command, that `print` command will be executed first. As written, it puts '`Executing`' and then the contents of the command line (see below) into the title of your xterm window, assuming it understands the usual xterm escape sequences. In fact, this particular example is usually handled with the special shell function (not zle function) '`preexec`' which is passed a command line about to be executed as an argument instead of in `$BUFFER`. There seems to be a side effect of rebinding `accept-line` that the return key stops working in the minibuffer under some circumstances.

Note that to undo the fact that return executes your new widget, you need to alias `accept-line` back to `.accept-line`:

```
zle -A .accept-line accept-line
```

If you have trouble remembering the order, as with most alias or rename commands in zsh and UNIX generally, including `ln` and `bindkey -A`, the existing command, the one whose properties you want to keep, comes first, while the new name for it comes second. Also, as with those commands, it doesn't matter if the second name on the line currently means something else; that will be replaced by the new meaning. Afterwards, you don't need to worry about your own `accept-line` widget; zle handles the details of removing widgets when they're no longer referred to. The function's still there, however, since as far as the rest of the shell is concerned it's just an ordinary shell function which you need to '`unfunction`' to remove.

Do remember, however, not to delete a widget which redefines a basic internal widget by the obvious command

```
# Noooo!
zle -D accept-line
```

which stops the return key having any effect other than complaining there's no such widget. If you get into real trouble, '`\ex.accept-line`' should work, as you can use the '`.`'-widgets anywhere you can use any other except where they would redefine or delete a '`.`' widget. Use the '`zle -A`' command above with the extended-command form of '`.accept-line`' to return to normality. If you try to redefine or delete a '`.`' widget, zle will tell you it's protected. You can remove any other widget in this way, however, even if it is still bound to a key sequence; you will then see an error if you type that sequence.

One point to note about `accept-line` is that the line isn't passed up to zsh instantly, only when your own function exits. This is pretty obvious when you think about it; zle is called from the main shell, and if your own zle widget hasn't finished executing, the main shell hasn't got control back yet. But it does mean, for example, that if you modify the command line after a call to `accept-line` or `.accept-line`, those changes are reflected in the line passed up to the shell:

```
# Noooo! to this one too.
accept-line() {
  zle .accept-line
  BUFFER='Ha ha!'
}
```

This always returns the string '`Ha ha!`' to the main shell. This is not particularly useful unless you are constructing a Samuel Beckett shell for display at an installation in a Parisian art gallery.

4.7.4 Special parameters: normal text

The shell makes various parameters available for easy manipulation of the command line. You've already seen `$NUMERIC`. You may wonder what happens if you have your own parmeter called `$NUMERIC`; after all, it's a fairly simple string to use as a name. The good

news is you don't need to worry; when the shell runs a zle function, it simply hides any existing occurrences of a parameter and makes its special parameters available. Then when it exits, the original parameter is reenabled. So all you have to worry about is making sure you don't use these special parameters for anything else while you are inside a zle widget.

There are four particularly common zle parameters.

First, there are three ways of referring to the text on the command line: `$BUFFER` is the entire line as a string, `$LBUFFER` is the line left of the cursor position, and `$RBUFFER` is the line after it including the character under the cursor, so that the division is always at the point where the next inserted character would go. Any or all of these may be empty, and `$BUFFER` is always the string `$LBUFFER$RBUFFER`.

The necessary counterpart to these is `$CURSOR`, which is the cursor position with 1 being the first character. If you know how the shell handles substrings in parameter substitutions, you will be able to see that `$LBUFFER` is `$BUFFER[1,$CURSOR-1]`, while `$RBUFFER` is `$BUFFER[$CURSOR,-1]` (unless you are using the option `KSH_ARRAYS` for compatibility of indexes with ksh — this isn't recommended for implementing zle or completion widgets as it causes confusion with the ones supplied with the shell).

The really useful thing about these is that they are modifiable. If you modify `$LBUFFER` or `$RBUFFER`, then `$BUFFER` and `$CURSOR` will be modified appropriately; lengthening or shortening `$LBUFFER` increases or decreases `$CURSOR`. If you modify `$BUFFER`, you may need to set `$CURSOR` yourself as the shell can't tell for sure where the cursor should be. If you alter `$CURSOR`, characters will be moved between `$LBUFFER` and `$RBUFFER`, but `$BUFFER` will remain the same.

This makes tasks along the lines of basic movement and deletion commands extremely simple, often just a matter of pattern matching. However, it definitely pays to know about zsh's more sophisticated pattern matching and parameter substitution features, described in the next chapter. For example, if you start a widget function with

```
emulate -L zsh
setopt extendedglob
LBUFFER=${LBUFFER%%[^[:blank:]]##}
```

then `$LBUFFER` contains the line left of the cursor stripped of all the non-blank characters (usually anything except space or tab) immediately to the left of the cursor.

This function uses the parameter substitution feature '`${`*param*`%%`*pattern*`}`' which removes the longest match of *pattern* from the end of `$`*param*. The '`emulate -L zsh`' ensures the shell options are set appropriately for the function and makes all option settings local, and '`setopt extendedglob`' which turns on the extended pattern matching features; it is this that makes the sequence '`##`' appearing in the pattern mean 'at least one repetition of the previous pattern element'. The previous pattern element is 'anything except a blank character'. Hence, all occurrences of non-blank characters are removed from the end of `$LBUFFER`.

If you want to move the cursor over those characters, you can tweak the function slightly:

```
emulate -L zsh
setopt extendedglob
chars=${(M)LBUFFER%%[^[:blank:]]##}
(( CURSOR -= ${#chars} ))
```

The string '(M)' has appeared at the start of the parameter substitution. This is part of zsh's unique system of parameter flags; this one means 'insert the matched portion of the substitution'. In other words, instead of returning $LBUFFER stripped of non-blank characters at the end, the substitution returns those very characters which it would have stripped. To skip over them is now a simple matter of decreasing $CURSOR by the length of that string.

You'll find if you try these examples that they probably don't do quite what you want. In particular, they don't handle any blank characters found next to the non-blank ones which normal word-orientated functions do. However, you now have enough information to add tests for that yourself.

If you get more sophisticated, you can then add handling for $NUMERIC. Remember this isn't set unless the user gave it explicitly, so it's up to you to treat it as 1 in that case.

4.7.5 Other special parameters

A large fraction of what you are likely to want to do can be done with the parameters we've already met. Here are some hints as to how you might want to use some of the other parameters available. As always, for a complete list with rather less in the way of hints see the manual.

$KEYS tells you the keys which were used to call the widget; it's a string of those raw characters, not turned into the bindkey format. In other words, if it was a single key (including possibly a control key or a meta key), $KEYS will just contain a single character. So you can change the widget's behaviour for different keys. Here's a very (very) simple function like self-insert:

```
LBUFFER=$LBUFFER$KEYS
```

Note this doesn't work very well with \ex extended command handling; you just get the ^m from the end of the line. You need to make sure any widgets which use $KEYS are sensibly bound. This also doesn't handle numeric arguments to repeat characters; it's a fairly simple exercise (particularly given zsh's 'repeat' loop) to add that.

$WIDGET and $LASTWIDGET tell you the name of the current widget being executed and the one before that. These don't sound all that useful at first hearing. However, you can use $WIDGET together with the fact that a widget doesn't need to have the same name as the function that defines it. You can define

```
zle -N this-widget function
zle -N that-widget function
```

and test $WIDGET inside function to see if it contains this-widget or that-widget. If these have a lot of shared code, that is a considerable simplification without having to write extra functions.

$LASTWIDGET tends to be used for a slightly different purpose: checking whether the last command to be executed was the same as the current one, or maybe was just friendly with it. Here are edited highlights of the function up-line-or-beginning-search, a sort of cross between up-line-or-search and history-beginning-search-backward which has been added to the shell distribution for 4.1. If there are previous lines in the buffer, it moves up through them; else if it's the first in a sequence of calls to this function it remembers

the cursor position and looks backwards for a line with the same text from the start up to that point, and puts the cursor at the end of the line; else if the same widget has just been executed, it uses the old cursor position to search for another match further back in the history.

```
if [[ $LBUFFER == *$'\n'* ]]; then
  zle .up-line-or-history
  __searching=''
else
  if [[ $LASTWIDGET = $__searching ]]; then
    CURSOR=$__savecursor
  else
    __savecursor=$CURSOR
  fi
  __searching=$WIDGET
  zle .history-beginning-search-backward
  zle .end-of-line
fi
```

We test `$__searching` instead of `$WIDGET` directly to be able to tell the case when we are moving lines instead of searching. `$__savecursor` gives the position for the backward search, after which we put the cursor at the end of the line. The parameters beginning '__' aren't local to the function, because we need to test them from the previous execution, so they have been given underscores in front to try to distinguish them from other parameters which might be around.

You'll see that the actual function supplied in the distribution is a little more complicated than this; for one thing, it uses styles set by the user to decide it's behaviour. Styles are described for use with completion widgets in chapter 6, but you can use them exactly the same way in zle functions.

The full version of `up-line-or-beginning-search` uses another parameter, `$PREBUFFER`. This contains any text already absorbed by `zle` which you can no longer edit — in other words, text read in before the shell prompted with `$PS2` for the remainder. Testing '`[[ -n $PREBUFFER ]]`' therefore effectively tests whether you are at the `$PS2`. You can use this to implement behaviour after the fashion of `push-line-or-edit`.

4.7.6 Reading keys and using the minibuffer

Every now and then you want the editor to do a sequence of operations with user input in the middle. This is usually done by a combination of two commands.

First, you may need to prompt the user in the minibuffer, just like `\ex` does. You can do this with '`zle -R`'. Its basic function is to redisplay the command line, flushing all the changes you have made in your function so far, but you can give it a string argument which appears in the minibuffer, just below the command line. You can give it a list of other strings after that, which appear in a similar way to lists of possible completions, but have no special significance to zle in this case.

To get input back from the user, you can use '`read -k`' which reads a single key (not a sequence; no lookup takes place). This command is always available in the shell, but in this case it is handled by zle itself. The key is returned as a raw byte. Two facilities of arithmetic evaluation are useful for handling this key: '`#key`' returns the ASCII code for the

first character of `$key`, while '`##`*key*' returns the ASCII code for *key*, which is in the form that `bindkey` would understand. For example,

```
read -k key
if (( #key == ##\C-g )); then
   ...
```

makes the use of arithmetic evaluation. The form on the left turns the first character in `$key` into a number, the second turns the literal bindkey-style string `\C-g` into a number (ASCII 7, since 1 to 26 are just `\C-a` to `\C-z`). Don't confuse either of these forms with '`$#key`', which is the length of the string in the parameter, in this case almost certainly 1 for a single byte; this form works both inside and outside arithmetic substitution, the other forms only inside. The '`(( ... ))`' form is recommended for arithmetic substitutions whenever possibly; you can do it with the basic '`[[ ... ]]`' form, since '`-eq`' and similar tests treat both sides as arithmetic, though you may need extra quoting; however, the only good reason I know for doing that is to avoid using two types of condition syntax in the same complex test.

These tricks are only really useful for quite complicated functions. For an example, look at the function `incremental-complete-word` supplied with the zsh source distribution. This function doesn't add to clarity by using the form '`#\\C-g`' instead of '`##\C-g`'; it does the same thing but the double backslash is very confusing, which is why the other form was introduced.

4.7.7 Examples

transpose-words-about-point

This function is a variant on `transpose-words`. It has various twists. First, the words in question are always space-delimited, neither shell words nor words in the `$WORDCHARS` sense. This makes it fairly predictable.

Second, it will transpose words about the current point (hence its name) even if the character under the cursor is not a whitespace character. I find this useful because I am eternally typing compound words a bit like '`function_name`' only to find that what I should have typed was '`name_function`'. Now I just position the cursor over the underscore and execute this widget.

```
emulate -L zsh
setopt extendedglob

local match mbegin mend pat1 pat2 word1 word2 ws1 ws2

pat1=${LBUFFER%%(#b)([^[:blank:]]##)([[:blank:]]#)}
word1=$match[1]
ws1=$match[2]

match=()
pat2=${RBUFFER##(#b)(?[[:blank:]]#)([^[:blank:]]##)}
ws2=$match[1]
word2=$match[2]

if [[ -n $word1 && -n $word2 ]]; then
```

```
    LBUFFER="$pat1$word2$ws1"
    RBUFFER="$ws2$word1$pat2"
  else
    zle beep
  fi
```

The only clever stuff here is the pattern matching. It makes a great deal of use of 'backreferences' an extended globbing feature which is used in all forms of pattern matching including, as in this case, parameter substitution. It will be described fully in the next chapter. The key things to look for are the '`(#b)`', which activates backreferences if the option `EXTENDED_GLOB` is turned on, the parentheses following that, which mark out the bits you want to refer to, and the references to elements of the array `$match`, which store those bits. The shell also sets `$mbegin` and `$mend` to give the positions of the start and end of those matches, which is why those parameters are made local; we want to preserve them from being seen outside the function even though we don't actually use them.

You might also need to know about the '`#`' characters: one after a pattern means 'zero or more repetitions', and two mean 'one or more repetitions'. Finally, '`[:blank:]`' in a character class refers to any blank character; when negated, as in the character class '`[^[:blank:]]`', it means any non-blank character. With the '`#`'s we match a series blank or non-blank characters. Given that, you can work out the rest of what's going on.

Here's a more sophisticated version of that. If you found the previous one heavy going. you probably don't want to look too closely at this.

```
emulate -L zsh
setopt extendedglob

local wordstyle blankpat wordpat1 wordpat2
local match mbegin mend pat1 pat2 word1 word2 ws1 ws2

zstyle -s ':zle:transpose-words-about-point' word-style wordstyle

case $wordstyle in
  (shell) local bufwords
          # This splits the line into words as the shell understands them.
          bufwords=(${(z)LBUFFER})
          wordpat1="${(q)bufwords[-1]}"
          # Take substring of RBUFFER to skip over first character,
          # which is the one under the cursor.
          bufwords=(${(z)RBUFFER[2,-1]})
          wordpat2="${(q)bufwords[1]}"
          blankpat='[[:blank:]]#'
          ;;
  (space) blankpat='[[:blank:]]#'
          wordpat1='[^[:blank:]]##'
          wordpat2=$wordpat1
          ;;
  (*) local wc=$WORDCHARS
      if [[ $wc = (#b)(?*)-(*) ]]; then
        # We need to bring any `-' to the front to avoid confusing
        # character classes... we get away with `]' since in zsh
        # this isn't a pattern character if it's quoted.
        wc=-$match[1]$match[2]
```

```
        fi
        # A blank is anything not in the character class consisting
        # of alphanumerics and the characters in $wc.
        # Quote $wc where necessary, because we don't want those
        # characters to be considered as pattern characters later on.
        blankpat="[^${(q)wc}a-zA-Z0-9]#"
        # and a word character is anything else.
        wordpat1="[${(q)wc}a-zA-Z0-9]##"
        wordpat2=$wordpat1
        ;;
esac

# The eval makes any special characters in the parameters active.
# In particular, we need the surrounding '[' s to be 'real'.
# This is why we quoted the wordpats in the 'shell' option, where
# they have to be treated as literal strings at this point.
eval pat1='${LBUFFER%%(#b)('${wordpat1}')('${blankpat}')}'
word1=$match[1]
ws1=$match[2]

match=()
eval pat2='${RBUFFER##(#b)(?'${blankpat}')('${wordpat2}')}'
ws2=$match[1]
word2=$match[2]

if [[ -n $word1 && -n $word2 ]]; then
  LBUFFER="$pat1$word2$ws1"
  RBUFFER="$ws2$word1$pat2"
else
  zle beep
fi
```

What has been added is the ability to use a style to define how the shell finds a 'word'. By default, words are the same as what the shell usually thinks of as a word; this is handled by the branch of the case statement which uses '`$WORDCHARS`' and a little extra trickery to get a pattern which matches the set of characters considered parts of a word. We used the `eval`'s because it allowed us to have some bits of `$wordpat1` and friends active as pattern characters while others were quoted.

This introduces two types of parameter expansion flags: `${(q)`*param*`}` adds backslashes to quote special characters in `$`*param*, so that when the parameter appears after `eval` the result is just the original string. `${(z)`*param*`}` splits the parameter just as if it were a shell command line being split into a command and words, so the result is an array; '`z`' stands for zsh-splitting or just zplitting as you fancy.

If you set

```
zstyle ':zle:*' word-style space
```

you get back to the behaviour of the original function.

Finally, if you replace '`space`' with '`shell`' in that `zstyle` command, you will get words as they are split for normal use within the shell; for example try

```
echo execute the widget 'between these' 'two quoted expressions'
```

and the entire quoted expressions will be transposed. You may find that if you do this in the middle of a quoted expression, you don't get a sensible result; that's because the `(z)`-splitting doesn't know what to do with the improperly completed quotes to its left and right. Some versions of the shell have a bug (fixed in 4.0.5) that the expressions which couldn't be split properly, because the quotes weren't complete, have an extra space character at the end.

insert-numeric

Here's a widget which allows you to insert an ASCII character which you know by number. I can't for the life of me remember where it came from, but it's been lying around apparently for two and a half years (please do email me if you think you wrote it, otherwise I'll assume I did). You can give it a numeric prefix (that's the easy part of the function), else it will prompt you for a number. If you type '`x`' or '`o`' the number is treated as hexadecimal or octal, respectively, else as decimal.

```
# Set up standard options.
# Important for portability.
emulate -L zsh

# x must display in hexadecimal
typeset -i 16 x

if (( ${+NUMERIC} )); then
  # Numeric prefix given; just use that.
  x=$NUMERIC
else
  # We need to read the ASCII code.
  local msg modes key mode=dec code char
  # Prompt for and read a base.
  integer base=10
  zle -R "ASCII code (o -> oct, x -> hex) [$mode]: "
  read -k key
  case $key in
    (o) base=8 mode=oct
        zle -R "ASCII code [$mode]: "
        read -k key
        ;;
    (x) base=16 mode=hex
        zle -R "ASCII code [$mode]: "
        read -k key
       ;;
  esac
  # Now we are looking for numbers in that base.
  # Loop until newline or return.
  while [[ '#key' -ne '##\n' && '#key' -ne '##\r' ]]; do
    if [[ '#key' -eq '##^?' || '#key' -eq '##^h' ]]; then
      # Delete a character
      [[ -n $code ]] && code=${code[1,-2]}
    elif [[ ($mode == hex && $key != [0-9a-fA-f]) ||
          ($mode == dec && $key != [0-9]) ||
```

```
            ($mode == oct && $key != [0-7]) ]]; then
        # Character not in range, beep
        zle beep
      elif [[ '#key' -eq '##\C-g' ]]; then
        # Abort: returning 1 signals to zle that this
        # is an abnormal termination.
        return 1
      else
        code="${code}${key}"
      fi
      char=
      if [[ -n $code ]]; then
        # Work out the character using the
        # numbers typed so far.
        (( x = ${base}#${code} ))
        if (( x > 255 )); then
          zle beep
          code=${code[1,-2]}
          [[ -n $code ]] && (( x = ${base}#${code} ))
        fi
        [[ -n $code ]] && eval char=\$\'\\x${x##???}\'
      fi

      # Prompt for any more digits, showing
      # the character as it would be inserted.
      zle -R "ASCII code [$mode]: $code${char:+ = $char}"
      read -k key || return 1
    done
    # If aborted with no code, return
    [[ -z $code ]] && return 0
    # Now we have the ASCII code.
    (( x = ${base}#${code} ))
  fi

  # Finally, if we have a single-byte character,
  # insert it to the left of the cursor
  if (( x < 0 || x > 255 )); then
    return 1
  else
    eval LBUFFER=\$LBUFFER\$\'\\x${x##???}\'
  fi
```

This shows how to do interactive input. The '`zle -R`'s prompt the user, while the '`read -k`'s accept a character at a time. As an extra feature, while you are typing the number, the character that would be inserted if you hit return is shown. The widget also handles deletion with backspace or the (UNIX-style, not PC-style) delete key.

One blight on this is the way of turning the number in x into a character, which is done by all those `eval`s and backslashes. It uses the feature that e.g. `$'\x41'` is the character `0x41` (an ASCII 'A'). To use this, we must make sure the character (stored in x) appears as hexadecimal, and following ksh zsh outputs hexadecimal numbers as '`16#41`' or similar. (The new option `C_BASES` shows hexadecimal numbers as 0x41 and similar, but here we need the plain number in any case.) Hence we strip the '`16#`' and construct our `$'\x41'`. Now we need to persuade the shell to interpret this as a quoted string by passing it to `eval` with the

special characters (`$`, `\`, `'`) quoted with a backslash so that they aren't interpreted too early.

By the way, note that zsh only handles ordinary 8-bit characters at the moment. It doesn't matter if some do-gooder on your system has set things up to use UTF-8 (a UNIX-friendly version of the international standard for multi-byte characters, Unicode) to appeal to the international market, I'm afraid zsh is stuck with ISO 8859 and similar character sets for now.

Chapter 5

Substitutions

This chapter will appeal above all to people who are excited by the fact that

```
print ${array[(r)${(l.${#${(O@)array//?/X}[1]}..?.)}]}
```

prints out the longest element of the array `$array`. For the overwhelming majority that forms the rest of the population, however, there should be plenty that is useful before we reach that stage. Anyway, it should be immediately apparent why there is no obfuscated zsh code competition.

For those who don't do a lot of function writing and spend most of the time at the shell prompt, the most useful section of this chapter is probably that on filename generation (i.e. globbing) at the end of the chapter. This will teach you how to avoid wasting your time with `find` and the like when you want to select files for a command.

5.1 Quoting

I've been using quotes of some sort throughout this guide, but I've never gone into the detail. It's about time I did, since using quotes is an important part of controlling the effects of the shell's various substitutions. Here are the basic quoting types.

5.1.1 Backslashes

The main point to make about backslashes is that they are really trivial. You can quote any character whatsoever from the shell with a backslash, even if it didn't mean anything unquoted; so if the worst comes to the worst, you can take any old string at all, whatever it has in it — random collections of quotes, backslashes, unprintable characters — quote every single character with a backslash, and the shell will treat it as a plain string:

```
print \T\h\i\s\ \i\s\ \*\p\o\i\n\t\l\e\s\s\*\ \
    \-\ \b\u\t\ \v\a\l\i\d\!
```

Remember, too that, this means you need an extra layer of quotation to pass a '`\n`', or whatever, down to `print`.

However, zsh has an easier way of making sure everything is quoted with a backslash when that's needed. It's a special form of parameter substitution, just one of many tricks you can do by supplying flags in parentheses:

```
% read string
This is a *string* with various 'special' characters
% print -r -- ${(q)string}
This\ is\ a\ \*string\*\ with\ various\ \'special\'\ characters
```

The `read` builtin didn't do anything to what you typed, so `$string` contains just those characters. The `-r` flag to print told it to print out what came after it in raw fashion, and here's the special part: `${(q)string}` tells the shell to output the parameter with backslashes where needed to prevent special characters being interpreted. All parameter flags are specific to zsh; no other shell has them.

The flag is not very useful there, because zsh usually (remember the `GLOB_SUBST` option?) doesn't do anything special to characters from substitutions anyway. Where it *is* extremely useful is if you are going to re-evaluate the text in the substitution but still want it treated as a plain string. So after the above,

```
% eval print -r -- ${(q)string}
This is a *string* with various 'special' characters
```

and you get back what you started with, because at the `eval` of the command line the backslashes put in by the `(q)` flag meant that the value was treated as a plain string.

You can strip off quotes in parameters, too; the flag `(Q)` does this. It doesn't care whether backslashes or single or double quotes are used, it treats them all the way the shell's parser would. You only need this when the parameter has somehow acquired quotes in its value. One way this can happen is if you try reading a file containing shell commands, and for this there's another trick: the `(z)` flag splits a line into an array in the same way as if the line had been read in and was, say, being assigned to an array. Here's an example:

```
% cat file
print 'a quoted string' and\ another\ argument
% read -r line <file
% for word in ${(z)line}; do
for> print -r "quoted:    $word"
for> print -r "unquoted:  ${(Q)word}"
for> done
quoted:    print
unquoted:  print
quoted:    'a quoted string'
unquoted:  a quoted string
quoted:    and\ another\ argument
unquoted:  and another argument
```

You will notice that the `(z)` doesn't remove any of the quotes from the words read in, but the `(Q)` flag does. Note the `-r` flags to both `read` and `print`: the first prevents the backslashes being absorbed by `read`, and the second prevents them being absorbed by `print`. I'm afraid backslashes can be a bit of a pain in the neck.

5.1.2 Single quotes

The only thing you can't quote with single quotes is another single quote. However, there's an option RC_QUOTES, where two single quotes inside a single-quoted string are turned into one. Apparently 'RC' refers to the shell rc which appeared in plan9; it seems to be one of those programmes that some people get fanatically worked up about while the rest of us can't quite work out why. Zsh users may sympathise. (This was corrected by Oliver Kiddle and Bart Schaefer after I guessed incorrectly that RC stood for recursive, although you're welcome to think of it that way anyway. It doesn't really work for RC_EXPAND_PARAM, however, which is definitely from the rc shell, and if you look at the source code you will find a variable called 'plan9' which is tested to see if that option is in effect.)

You might remember something like this from BASIC, although in that case with double quotes — in zsh, it works only with single quotes, for some reason. So,

```
print -r 'A ''quoted'' string'
```

would usually give you the output 'A quoted string', but with the option set it prints 'A 'quoted' string'. The -r option to print doesn't do anything here, it's just to show I'm not hiding anything. This is usually a useful and harmless option to have set, since there's no other good reason for having two quotes together within quotes.

The standard way of quoting single quotes is to end the quote, insert a backslashed single quote, and restart quotes again:

```
print -r 'A '\''quoted'\'' string'
```

which is unaffected by the option setting, since the quotes immediately after the backslashes are always treated as an ordinary printable character. What you *can't* ever do is use backslashes as a way of quoting characters inside single quotes; they are just treated as ordinary characters there.

You can make parameter flags produce strings quoted with single quotes instead of backslashes by doubling the 'q': '${(qq)param}' instead of '${(q)param}'. The main use for this is that the result is shorter if you know there are a lot of special characters in the string, and it's also a bit more easy to read for humans rather than machines, but usually it gains nothing over the other form. It can tell whether you have RC_QUOTES set and uses that to make the string even shorter, so be careful if you might use the resulting string somewhere where the option isn't set.

5.1.3 POSIX quotes

There's a relative of single quotes which uses the syntax $' to introduce a quoted string and ' to end it; I refer to them as 'POSIX quotes' because they appear in the POSIX standard and I don't know what else to call them; 'string quotes' is one possibility, but sounds a bit vague (what else would you quote?) The difference from single quotes is that they understand the same backslash sequences as the print builtin. Hence you can have the convenience of using '\n' for newline, '\e' for escape, '\xFF' for an arbitrary character in hexadecimal, and so on, for any command:

```
% cat <<<$'Line\tone\nLine\ttwo'
```

```
Line    one
Line    two
```

Remember the 'here string' notation '«<', which supplies standard input for the command. Hence the output shows exactly how the quoted string is being interpreted. It is the same as

```
% print 'Line\tone\n\Line\ttwo'
Line    one
Line    two
```

but there the interpretation is done inside `print`, which isn't always convenient. POSIX quotes are currently rather underused.

This is as good a point as any to mention that the shell is completely 'eight-bit clean', which means you can have any of the 256 possible characters anywhere in your string. For example, `$'foo\000bar'` has an embedded ASCII NUL in it (that's not a misprint — officially, ASCII non-printing characters have two- or three-letter abbreviations). Usually this terminates a string, but the shell works around this when you are using it internally; when you try and pass it as an argument to an external programme, however, all bets are off. Almost certainly the first NUL in that case will cause the programme to think the string is finished, because no information about the length of arguments is passed down and there's nothing the shell can do about it. Hence, for example:

```
% echo $'foo\000bar'
foobar
% /bin/echo $'foo\000bar'
foo
```

The shell's `echo` knows about the shell's 8-bit conventions, and prints out the NUL, which the terminal doesn't show, then the remainder of the string. The external version of `echo` didn't know any better than to stop when it reached the NUL.

There are actually uses for embedded NULs: some versions of `find` and `xargs`, for example, will put or accept NULs instead of newlines between their bits of input and output (as distinct from command line arguments), which is much safer if there's a chance the input or output can contain a live newline. Using `$'\000'` allows the shell to fit in very comfortably with these. If you want to try this, the corresponding options are `-print0` for `find` (print with a NUL terminator instead of newline) and `-0` for `xargs` (read input assuming a NUL terminator).

In older versions of the shell, characters with the top bit set, such as those from non-English character sets found in ISO 8859 fonts, could cause problems, since the shell also uses such characters internally to represent its own special characters, but recent versions of the shell (from about 3.0) side-step this problem in the same way as for NULs. Any remaining problems — it's quite tricky to handle this completely consistently — are bugs and should be reported.

You can force parameters to be quoted with POSIX quotes by the somewhat absurd expedient of making the `q` in the quote flag appear a total of four times. I can't think why you would ever want to do that, except that it will turn newlines into '`\n`' and hence the result will fit on a single (maybe rather long) line. Plus you get the replacement of funny characters with escape sequences.

5.1.4 Double quotes

Double quotes allow some, but not all, forms of substitution inside. More specifically, they allow parameter expansion, command substitution and arithmetic substitution, but not any of the others: process substitution doesn't happen, braces and initial tildes and equals signs are not expanded and patterns are not special. Here's a table; each expression on the left is some command line argument, and the results show what is substituted if it appears outside quotes, or in double quotes.

```
Expression      Outside quotes  In double quotes
-----------------------------------------------
=(echo hi mum)  /tmp/zshTiqpL     =(echo hi mum)
$ZSH_VERSION    4.0.1             4.0.1
$(echo hi mum)  hi mum            hi mum
$((6**2 + 6))   42                42
{a,b}cd         acd bcd           {a,b}cd
~/foo           /home/pws/foo     ~/foo
.zl*            .zlogin .zlogout  .zl*
```

That '`/tmp/zshTiqpL`' could be any temporary filename, and indeed several of the other substitutions will be different in your case.

You might already have guessed that '`${(qqq)string}`' forces `$string` to use double quotes to quote its special characters. As with the other forms, this is all properly handled — the shell knows just which characters need quoting inside double quotes, and which don't.

Word-splitting in double quotes

Where the substitutions are allowed, the (almost) invariable side effect of double quotes is that word-splitting is suppressed. You can see this using '`print -l`', which prints one argument per line:

```
% array=(one two)
% print -l $(echo foo bar) $array
foo
bar
one
two
% print -l "$(echo foo bar) $array"
foo bar one two
```

The reason this is 'almost' invariable is that parameter substitution allows you to specify that normal word-splitting will occur. There are two ways of doing this; both use the symbol '`@`'. You probably remember this from the parameter '`$@`' which has just that effect when it appears in double quotes: the arguments to the script or function are split into words like a normal array, except that empty arguments are not removed. I covered this at some length in chapter 3.

This is extended for other parameters in the following way:

```
% array=(one two three)
```

```
% print -l "${array[@]}"
one
two
three
```

and more generally for all forms of substitution using another flag, `(@)`:

```
% print -l "${(@)array}"
one
two
three
```

Digression on subscripts

The version with flags is perhaps less clear than the other, but it can appear in lots of different places. For example, here is how you pick a slice of an array in zsh:

```
% print -l ${array[2,-1]}
two
three
```

where negative numbers count from the end of the array. The numbers in square brackets are referred to as subscripts. This can get the `(@)` treatment, too:

```
% print -l "${(@)array[2,-1]}"
two
three
```

Although it's probably not obvious, you can use the other notation in this case:

```
% print -l "${array[@][2,-1]}"
two
three
```

The shell will actually handle arbitrary numbers of subscripts in parameter substitutions, not just one; each applies to the result of the previous one:

```
% print -l "${array[@][2,-1][1]}"
two
```

What you have to watch out for is that that last subscript selected a single word. You can continue to apply subscripts, but they will apply only on the *characters* in that word, not on array elements:

```
% print -l "${array[@][2,1][1][2,-1]}"
wo
```

We've now strayed severely off topic: the subscripts will of course work quite independently from whether the word is being split or appears in double quotes. Despite the joining of

words that occurs in double quotes, subscripts of arrays still select array elements. This is a consequence of the order in which the rules of parameter expansion apply. There is a long, involved section on this in the `zshexpn` manual entry (look for the heading 'Rules' there or in the 'Parameter Expansion' node of the corresponding Info or HTML file).

Word-splitting of quoted command substitutions

Zsh has the useful feature that you can force the shell to apply the rules of parameter expansion to the result of a command substitution. To see where that might be useful, consider the case of the special 'command substitution' (although it's handled entirely in the shell, not by running an external command) which puts the contents of a file on the command line:

```
% args() { print $#; }    # report number of arguments
% cat file
Words on line one
Words on line two
% args $(<file)
8
% args "$(<file)"
1
```

The unquoted substitution split the file into individual words; the quoted substitution didn't split it at all. These are the standard shell rules.

It's very common, however, that you want one line per argument, not splitting on spaces within the line. This is where parameter expansion can come in. There is a flag `(f)` which says 'split the result of the expansion, one word per line'. Here's how to use it in this case:

```
% args "${(f)$(<file)}"
2
```

Where you would usually put the name of a parameter, you put the command substitution instead, and the shell operates on the result of that (note that it does not treat the result as the name of a parameter, but as a value — this is discussed in more detail below). The double quotes were necessary because otherwise the file would already have been split into individual words by the time the parameter substitution came to look at the result. You can easily verify that the two arguments are the individual lines of the file. I don't remember what the '`f`' stands for, but we were already using up flag codes quite fast when it came along; Bart Schaefer believes it stands for 'fold', which might at least help you remember it.

5.1.5 Backquotes

The main thing to say about backquotes is that you should use the other form of command substitution instead. There are two good reasons.

First, the other form can be nested:

```
% print $(print $(print a word))
a word
```

Obviously that's a silly example, but the main point is that the only time parentheses should occur unquoted in the shell is in pairs (the patterns in case statements are an exception, but pairs of parentheses around patterns are valid, too, and I have used that form in this guide). Thus you can be confident that any piece of well-formatted shell code can appear inside the command substitution.

This is clearly not true with `` `...` ``, even though the basic effect is the same. Any unquoted `` ` `` which happens to appear in a chunk of code within the backquotes will be treated as the end of the quotes.

The second reason, which is closely related, is that it can be quite difficult to decide how many levels of quotes are required inside a backquoted expression. Consider:

```
% print "`echo \"hello\"`"
hello
% print "$(echo \"hello\")"
"hello"
```

It's hard to explain quite what the difference here is without waving my hands, which prevents me from typing, but the essential point is really the same one about nesting: you can't do it with backquotes, because the start and end symbols are the same, but you can do it with parentheses. So in the second case there is no doubt that the embedded command line, '`echo \"hello\"`', is to be treated exactly as if that had appeared outside the command substitution; whereas in the first place, the quotes within quotes had to be, um, quoted.

As a consequence, in

```
% print "$(echo "hello")"
hello
```

you need to be careful: at first glance, the pairs of double quotes surround '`$(echo `' and '`)`', but they don't, they are nested by virtue of the substitution. You see the same thing with parameter substitution:

```
% unset foo
% print "${foo:-"a string"}"
a string
```

A third, less good, reason for using the form with parentheses is that your more sophisticated friends will laugh at you otherwise. Peer pressure is so important in this complex world.

That's all I have to say about command substitution, since I already said a lot about it when I discussed the basic syntax in chapter 3.

5.2 Modifiers and what they modify

Modifiers were introduced in chapter 2 when I talked about 'bang history', since that's where they came from. In zsh, however, they can be used in a couple of other places. They have the same form in each case: a colon, followed by a letter which is the code for what the modifier does, possibly (in the case of substitutions) followed by some other string. So, to jog your memory, unless you have `NO_BANG_HIST` set:

```
% print ~/file
/home/pws/file
% print !-1:t
file
```

where ':t' takes the tail (non-directory part) of the filename.

The second use is in parameters. This follows on very naturally. Note that neither this nor any of the later uses of modifiers rely on the NO_BANG_HIST option; that's purely for history.

```
% param=~/file
% print ${param:t}
file
```

Normally you can miss out the braces in the parameter substitution, but I tend to use them with modifiers for the sake of clarity. The fact that the same parts of the shell are used for modifiers wherever they come from has certain consequences:

```
% print foo
foo
% ^foo^bar
bar
% param='this sentence contains a foo.'
% print ${param:&}
this sentence contains a bar.
```

The ampersand repeats the last substitution, which is the same for parameter modifiers as for history modifiers. I find parameter modifiers even more useful than history ones; extracting the head or tail of a path is a very common operation on parameters.

Modifiers are also smart enough to handle arrays in a useful fashion. Note this is not true of sets of arguments in history expansions; ':t' will only extract one tail in that case, which may not be quite what you're expecting:

```
% print a sentence with a /real/live/bogus/path in it.
% print !!:t
path in it.
```

However, arrays *are* handled the way you might hope:

```
% array=(~/.zshenv ~/.zshrc ~/.zlogout)
% print ${array:t}
.zshenv .zshrc .zlogout
```

The same logic is applied with substitutions. This means that the first match in every element of the array is replaced:

```
% array=('a bar of chocolate' 'a bar of barflies'
array> 'a barrier of barns')
% print ${array:s/bar/car/}
a car of chocolate a car of barflies a carrier of barns
```

unless, of course, you do a global replacement:

```
% print ${array:gs/bar/car/}
a car of chocolate a car of carflies a carrier of carns
```

Note, however, that parameter substitution has its own *much* more powerful equivalent, which does pattern matching, partial replacement of modified parts of the original string, and so on. We'll come to this all in good time.

The final use of modifiers is in filename generation, i.e. globbing. Since this usually works by having special characters on the command line, and modifiers just consist of ordinary characters, the syntax is a little different:

```
% print *.c
parser.c lexer.c input.c output.c
% print *.c(:r)
parser lexer input output
```

so you need parentheses around them. This is a special case of 'glob qualifiers' which you'll meet below; you can mix them, but the modifiers must appear at the end. For example,

```
% print -l ~/stuff/*
/home/pws/stuff/onefile.c
/home/pws/stuff/twofile.c
/home/pws/stuff/subdir
% print ~/stuff/*(.:r:t)
onefile twofile
```

The globbing qualifier '`.`' specifies that files must be regular, i.e. not directories nor some form of special file. The '`:r`' removes the suffix from the result, and the '`:t`' takes away the directory part. Consequently, filename modifiers will be turned off if you set the option `NO_BARE_GLOB_QUAL`.

Two final points to note about modifiers with filenames. First, it is the only form of globbing where the result is no longer a filename; it is always performed right at the end, after all normal filename generation. Presumably, in the examples above, the word which was inserted into the command line doesn't actually correspond to a real file any more.

Second, although it *does* work if the word on the command line isn't a pattern but an ordinary word with a modifier tacked on, it *doesn't* work if that pattern, before modification, doesn't correspond to a real file. So '`foo.c(:r)`' will only strip off the suffix if `foo.c` is there in the current directory. This is perfectly logical given that the attempt to match a file kicks the globbing system, including modifiers, into action. If this is a problem for you, there are ways round; for example, insert the right value by hand in a simple case like this, or more realistically store the value in a parameter and apply the modifier to that.

5.3 Process Substitution

I don't have much new to say on process substitution, but I do have an example of where I find it useful. If you use the pager 'less' you may know it has the facility to preprocess the files you look at, for example uncompressing files temporarily via the environment variable

$LESSOPEN (and maybe $LESSCLOSE). Zsh can very easily and, to my thoroughly unbiased way of looking, more conveniently do the same thing. Here's a subset of my zsh function front-end to less — or indeed any pager, which is given here by the standard environment variable $PAGER with the default less. You can hard-wire any file-displaying command at that point if you prefer.

```
integer i=1
local args arg
args=($*)

for arg in $*; do
  case $arg in
    (*.bz2) args[$i]="=(bunzip2 -c ${(q)arg})"
            ;;
    # this assumes your zcat is the one installed with gzip:
    (*.(gz|Z)) args[$i]="=(zcat ${(q)arg})"
               ;;
    (*) args=${(q)arg}
        ;;
  esac
  (( i++ ))
done

eval command ${PAGER:-less} $args
```

The main pieces of interest is how elements of the array $args were replaced. The reason each argument was given an extra layer of quotes via (q) is the eval at the end; $args is turned into an array of literal characters first, which hence need quoting to protect special characters. Without that, filenames with spaces or asterisks or whatever wouldn't be shown properly.

The reason the eval is there is so that the process substitutions are evaluated on the command line when the pager is run, and not before. They are assigned back to elements of $args in quotes, so don't get evaluated at that point. The effect will be to turn:

```
less file.gz file.txt
```

into

```
less =(zcat file.gz) file.txt
```

The 'command' at the end of the function is there just in case the function has the same name as the pager (i.e. 'less' in this example); it forces the external command to be called rather than the function. The process substitution is ideal in this context; it provides less with the name of a file to which the decompressed contents of file.gz have been sent, and it deletes the file after the command exits. Furthermore, the substitution happens in such a way that you can still specify multiple files on the command line as you usually can with less. The only problem is that the filename that appears in the 'less' prompt is meaningless.

In case you haven't come across it, bzip2 is a programme very similar to gzip, and it is used almost identically, but it provides better compression.

There's an infelicity in output process substitutions, just as there is with multios.

```
echo hello > >(sed s/hello/goodbye)
```

The shell spawns the `sed` process to handle the output from the command line — and then forgets about it. It does not wait for it (at least, not until after it exits, when it will use the `wait` system call to tidy up). So it is dangerous to rely on the result of the process being available in the next command. If you try it interactively, in fact, you may well find that the next prompt is printed before the output from `sed` shows up on the terminal. This can probably be considered a bug, but it is quite difficult to fix.

5.4 Parameter substitution

You can probably see from the above that parameter substitutions are at the heart of much of the power available to transform zsh command lines. What's more, we haven't covered even a significant fraction of what's on offer.

5.4.1 Using arrays

The array syntax in zsh is quite powerful (surprised?); just don't expect it to be as efficient as, say, perl. Like other features of zsh, it exists to make users' lives easier, not to make your computer run blindingly fast.

I've covered, somewhat sporadically, how to set arrays, and how to extract bits of them — the following illustrates this:

```
% array=(one two three four)
% print ${array}
one two three four
% print ${array[3]}
three
% print ${array[2,-1]}
two three four
```

Remember you need '`typeset`' or equivalent if you want the array to be local to a function. The neat way is '`typeset -a`', which creates an empty array, but as long as you assign to the array before trying to use it any old `typeset` will do.

You can use the array index and array slice notations for assigning to arrays, in other words on the left-hand side of an '=':

```
% array=(what kind of fool am i)
% array[2]=species
% print $array
what species of fool am i
% array[2]=(a piece)
% print $array
what a piece of fool am i
% array[-3,-1]=(work is a man)
% print $array
what a piece of work is a man
```

So you can replace a single element of an array by a single element, or by an array slice; likewise you can replace a slice in one go by a slice of a different length — only the bits you explicitly tell it to replace are changed, the rest is left intact and maybe shifted along to make way. This is similar to perl's 'splice' command, only for once maybe a bit more memorable. Note that you shouldn't supply any braces on the left hand side. The appearance of the expression in an assignment is enough to trigger the special behaviour of subscripts, even if `KSH_ARRAYS` is in effect — though you need to subtract one from your subscripts in that case.

You can remove bits in the middle, too, but note you should use an empty array:

```
% array=(one two three four)
% print $#array
4
% array[2]=
% print $#array
4
% array[2]=()
% print $#array
3
```

The first assignment set element 2 to the empty string, it didn't remove it. The second replaced the array element with an array of length zero, which did remove it.

Just as parameter substitutions have flags for special purposes, so do subscripts. You can force them to search through arrays, matching on the values. You can return the value matched ((r)everse subscripting):

```
% array=(se vuol ballare signor contino)
% print ${array[(r)s*]}
se
% print ${array[(R)s*]}
signor
```

The `(r)` flag takes a pattern and substitutes the first element of the array matched, while the `(R)` flag does the same but starting from the end of the array. If nothing matched, you get the empty string; as usual with parameters, this will be omitted if it's the only thing in an unquoted argument. Using our `args` function to count the arguments passed to a command again:

```
% array=(some words)
% args() { print $#; }
% args ${array[(r)s*]}
1
% args ${array[(r)X*]}
0
% args "${array[(r)X*]}"
1
```

where in the last case the empty string was quoted, and passed down as a single, empty argument.

You can also return the index matched; `(i)` to start matching from the beginning, and `(I)` to start from the end.

```
% array=(se vuol venire nella mia scuola)
% print ${array[(i)v*]}
2
% print ${array[(I)v*]}
3
```

matching 'vuol' the first time and 'venire' the second. What happens if they don't match may be a little unexpected, but is reasonably logical: you get the next index along. In other words, failing to match at the end gives you the length of the array plus one, and failing to match at the beginning gives you zero, so:

```
array=(three egregious words)
for pat in '*e*e*' '*a*a*'; do
  if [[ ${array[(i)$pat]} -le ${#array} ]]; then
    print "Pattern $pat matched in array: ${array[(r)$pat]}."
  else
    print "Pattern $pat failed to match in array"
  fi
done
```

prints:

```
Pattern *e*e* matched in array: three.
Pattern *a*a* failed to match in array
```

If you adapt that chunk of code, you'll see you get the indices 1 and 4 returned. Note that the characters in `$pat` were treated as a pattern even though putting `$pat` on the command line would normally just produce the characters themselves. Subscripts are special in that way; trying to keep the syntax under control at this point is a little hairy. There is a more detailed description of this in the manual in the section 'Subscript Parsing' of the `zshparam` manual page or the 'Array Parameters' info node; to quote the characters in `pat`, you would actually have to supply the command line strings `'\*e\*e\*'` and `'\*a\*a\*'`. Just go round mumbling 'extra layer of pattern expansion' and everyone will think you know what you're talking about (it works for me, fitfully).

There is currently no way of extracting a complete set of matches from an ordinary array with subscript flags. We'll see other ways of doing that below, however.

5.4.2 Using associative arrays

Look back at chapter 3 if you've forgotten about associative arrays. These take subscripts, like ordinary arrays do, but here the subscripts are arbitrary strings (or keys) associated with the value stored in the element of the array. Remember, you need to use '`typeset -A`' to create one, or one of `typeset`'s relatives with the same option. This means that if you created it inside a function it will be limited to the local scope, so if you want to create a global associative array you will need to give the `-g` flag as well. This is particularly common with associative arrays, which are often used to store global information such as configuration details.

Retrieving information from associative arrays can get you into some of the problems already hinted at in the use of subscript flags with arrays. However, since normal subscripting

doesn't make patterns active, there is a way round here: make the subscript into another parameter:

```
% typeset -A assoc
% assoc=(key value Shlüssel Wert clavis valor)
% subscript='key'
% print ${assoc[$subscript]}
value
```

I used fairly boring keys here, but they can be any string of characters:

```
% assoc=(']' right\ square\ bracket '*' asterisk '@' at\ sign)
% subscript=']'
% print ${assoc[$subscript]}
right square bracket
```

and *that* is harder to get the other way. Nonetheless, if you define your own keys you will often use simple words, and in that case they can happily appear directly in the square brackets.

I introduced two parameter flags, `(k)` and `(v)` in chapter 3:

```
% print ${(k)assoc}
* ] @
```

prints out keys, while

```
% print ${(kv)assoc}
* asterisk ] right square bracket @ at sign
```

and the remaining two possibilities do the same thing:

```
% print ${(v)assoc}
asterisk right square bracket at sign
% print ${assoc}
asterisk right square bracket at sign
```

You now know these are part of a much larger family of tricks to apply to substitutions. There's nothing to stop you combining flags:

```
% print -r ${(qkv)assoc}
\* asterisk \] right\ square\ bracket @ at\ sign
```

which helps see the wordbreaks. Don't forget the '`print -l`' trick for separating out different words, and hence elements of arrays and associative arrays:

```
% print -l ${(kv)assoc}
*
asterisk
]
```

```
right square bracket
@
at sign
```

which is quite a lot clearer. As always, this will fail if you engage in un-zsh activities with `SH_WORD_SPLIT`, but judicious use of `@`, whether as a flag or a subscript, and double quotes, will always work:

```
% print -l "${(@kv)assoc}"
*
asterisk
]
right square bracket
@
at sign
```

regardless of the option setting.

Apart from the subscripts, the second major difference between associative and ordinary arrays is that the former don't have any order defined. This will be entirely familiar if you have used Perl; the principle here is identical. However, zsh has no notion at all, even as a convenience, of slices of associative arrays. You can assign individual elements or whole associative arrays — remembering that in the second case the right hand side must consist of key/value pairs — but you can't assign subgroups. Any attempt to use the slice notation with commas will be met by a stern error message.

What zsh does have, however, is extra subscript flags for you to match and retrieve one or more elements. If instead of an ordinary subscript you use a subscript preceded by the flag `(i)`, the shell will search for a matching key (not value) with the pattern given and return that. This is deliberately the same as searching an ordinary array to get its key (which in that case is just a number, the index), but note this time it doesn't match on the value, it really does match, as well as return, the key:

```
% typeset -A assoc
% assoc=(fred third\ man finnbar slip roger gully trevor long\ off)
% print ${assoc[(i)f*]}
fred
```

You can still use the parameter flags `(k)` and `(v)` to tell the shell which part of the key and/or value to return:

```
% print ${(kv)assoc[(i)f*]}
fred third man
```

Note the division of labour. The subscript flag tells the shell what to match against, while the parameter flags tell it which bit of the matched element(s) you actually want to see.

Because of the essentially random ordering of associative arrays, you couldn't tell here whether fred or finnbar would be chosen. However, you can use the capital form `(I)` to tell the shell to retrieve all matches. This time, let's see the values of the elements for which the keys were matched:

```
% print -l ${(v)assoc[(I)f*]}
```

```
third man
slip
```

and here we also got the position occupied by `finnbar`. The same rules about patterns apply as with `(r)` in ordinary arrays — a subscript is treated as a pattern even if it came from a parameter substitution itself.

You probably aren't surprised to hear that the subscript flags `(r)` and `(R)` try to match the values of the associative array rather than its keys. These, too, print out the actual part matched, here the value, unless you use the parameter flags.

```
% print ${assoc[(r)*i*]}
third man
% print ${(k)assoc[(R)*i*]}
fred finnbar
```

There's one more pair of subscript flags of particular relevance to associative arrays, `(k)` and `(K)`. These work a bit like a case statement: the subscripts are treated as strings, and the keys of the associative arrays as patterns, instead of the other way around. With `(k)`, the value of the first key which matches the subscript is substituted; with `(K)`, the values of all matching keys are substituted

```
% typeset -A assoc
% assoc=('[0-9]' digit '[a-zA-Z]' letter '[^0-9a-zA-Z]' neither)
% print ${assoc[(k)0]}
digit
% print ${assoc[(k)_]}
neither
```

In case you're still confused, the '`0`' in the first subscript was taken as a string and all the keys in `$assoc` were treated as patterns in turn, a little like

```
case 0 in
  ([0-9]) print digit
          ;;
  ([a-zA-Z]) print letter
             ;;
  ([^0-9a-zA-Z]) print neither
                 ;;
esac
```

One important way in which this is *not* like the selection in a case statement is that you can't rely on the order of the comparison, so you can't rely on more general patterns being matched after more specific ones. You just have to use keys which are sufficiently explicit to match just the strings you want to match and no others. That's why we picked the pattern '`[^0-9a-zA-Z]`' instead of just '`*`' as we would probably have used in the case statement.

I said storing information about configuration was a common use of associative arrays, but the shell has a more powerful way of doing that: styles, which will figure prominently in the discussion of programmable completion in the next chapter. The major advantage of styles over associative arrays is that they can be made context-sensitive; you can easily make the same style return the same value globally, or make it have a default but with a different

value in one particular context, or give it a whole load of different values in different places. Each shell application can decide what is meant by a 'context'; you are not tied to the same scheme as the completion system uses, or anything like it. Use of hierarchical contexts in the manner of the completion system does mean that it is easy to create sets of styles for different modules which don't clash.

Here, finally, is a comparison of some of the uses of associative arrays in perl and zsh.

```
    perl                            zsh
-------------------------------------------------------------------
%hash = qw(key value);          typeset -A hash; hash=(key value)
$hash{key}                      ${hash[key]}
keys %hash                      ${(k)hash}
values %hash                    ${(v)hash}
%hash2 = %hash;                 typeset -A hash2; hash2=("${(@kv)hash}")
unset %hash;                    unset hash
if (exists $hash{key}) {        if (( ${+hash[key]} )); then
  ...                             ...
}                               fi
```

One final reminder: if you are creating associative arrays inside a function which need to last beyond the end of the function, you should create them with '`typeset -gA`' which puts them into the surrounding scope. The '`-g`' flag is of course useful with all types of parameter, but the associative array is the only type that doesn't automatically spring into existence when you assign to it in the right context; hence the flag is particularly worthy of note here.

5.4.3 Substituted substitutions, top- and tailing, etc.

There are many transformations which you can do on the result of a parameter substitution. The most powerful involve the use of patterns. For this, the more you know about patterns, the better, so I will reserve explanation of some of the whackiest until after I have gone into more detail on patterns. In particular, it's useful if you know how to tell the shell to mark subexpressions which it has matched for future extraction. However, you can do some very useful things with just the basic patterns common to all shells.

Standard forms: lengths

I'll separate out zsh-specific forms, and start off with some which appear in all shells derived from the Bourne shell. A more compact (read: terse) list is given in the manual, as always.

A few simple forms don't use patterns. First, the substitution `${#`*param*`}` outputs the length of `$`*param*. In zsh, you don't need the braces here, though in most other shells with this feature you do. Note that `${#}` on its own is the number of parameters in the command line argument array, which is why explicit use of braces is clearer.

`$#` works differently on scalar values and array values; in the former case, it gives the length in characters, and in the latter case the length in elements. Note that I said 'values', not 'parameters' — you have to work out whether the substitution is giving you a scalar or an array:

```
% print ${#path}
```

```
8
% print ${#path[1]}
13
```

The first result shows I have 8 directories in my path, the latter that the first directory (actually '/home/pws/bin') has 13 characters. You should bear this in mind with nested substitutions, as discussed below, which can also return either an array or a scalar.

Earlier versions of zsh always returned a character count if the expression was in double quotes, or anywhere the shell evalauted the expression as a single word, but that doesn't happen any more; it depends only on the type of the value. However, you can force the shell to count characters by using the (c) flag, and to count words (even in scalars, which it will split if necessary) by using (w):

```
% print ${#PATH}
84
% print ${(c)#path}
84
% foo="three scalar words"
% print ${(w)#foo}
3
```

Comparing the first two, you will see that character count with arrays includes the space used for separating (equal to the number of colons separating the elements in $PATH). There's a relative of (w) called (W), which treats multiple word separators as having zero-length words in between:

```
% foo="three  well-spaced  word"
% print ${(w)#foo}
3
% print ${(W)#foo}
5
```

giving two extra words over (w), which treats the groups of spaces in the same way as one. Being parameter flags, these modifications of the syntax are specific to zsh.

Note that if you use lengths in an arithmetic context (inside ((...)) or $((...))), you must include the leading '$', which you don't need for substituting the parameters themselves. That's because '#foo' means something different here — the number in the ASCII character set (or whatever extension of it you are using if it is an extended character set) of the first character in $foo.

Standard forms: conditional substitutions

The next group of substitutions is a whole series where the parameter is followed by an option colon and then '-', '=', '+' or '?'. The colon has the same effect in each case: without a colon, the shell tests whether the parameter is set before performing the operation, while with the colon it tests whether the parameter has non-zero length.

The simplest is '${*param*:-*value*}'. If $param has non-zero length (without the colon, if it is set at all), use its value, else use the *value* supplied. Suppose $foo wasn't set at the start of the following (however unlikely that may seem):

```
% print ${foo-bar}
bar
% foo=''
% print ${foo-bar}

% print ${foo:-bar}
bar
% foo='please no anything but bar'
% print ${foo:-bar}
please no anything but bar
```

It's more usual to use the form with the colon. One reason for that is that in functions you will often create the parameter with a `typeset` before using it, in which case it always exists, initially with zero length, so that the other form would never use the default value. I'll use the colon for describing the other three types.

'`${`*param*`:=`*value*`}`' is similar to the previous type. but in this case the shell will not only substitute *value* into the line, it will assign it to *param* if (and only if) it does so. This leads to the following common idiom in scripts and functions:

```
: ${MYPARAM:=default}  ${OTHERPARAM:=otherdefault}
```

If the user has already set `$MYPARAM`, nothing happens, otherwise it will be set to '`default`', and similarly for `${OTHERPARAM}`. The '`:`' command does nothing but return true after the command line has been processed.

'`${`*param*`:+`*value*`}`' is the opposite of '`:-`', logically enough: the *value* is substituted if the parameter *doesn't* have zero length. In this case, *value* will often be another parameter substitution:

```
print ${value:+"the value of value is $value"}
```

prints the string only if `$#value` is greater than zero. Note that what can appear after the '`+`' is pretty much any single word the shell can parse; all the usual single-word substitutions (so globbing is excluded) will be applied to it, and quotes will work just the same as usual. This applies to the values after '`:-`' and '`:=`', too. One other commonly seen trick might be worth mentioning:

```
print ${1+"$@"}
```

substitutes all the positional parameters as they were passed if the first one was set (here you don't want the colon). This was necessary in some old shells because `"$@"` on its own gave you a single empty argument instead of no arguments when no arguments were passed. This workaround isn't necessary in zsh, nor in most modern Bourne-derived shells. There's a bug in zsh's handling, however; see the section on function parameters in chapter 3.

The final type isn't that often used (meaning I never have): `${`*param*`?`*message*`}` tests if *param* is set (no colon), and if it isn't, prints the message and exits the shell. An interactive shell won't exit, but it will return you immediately to the prompt, skipping anything else stored up for execution. It's a rudimentary safety feature, a little bit like 'assert' in C programmes; most shell programmers seem to cover the case of missing parameter settings by more verbose tests. It's quite neat in short shell functions for interactive use:

```
mless() { mtype ${@:?missing filename} | $PAGER }
```

Standard forms: pattern removal

Most of the more sophisticated Bourne-like shells define two pairs of pattern operators, which I shall call 'top and tail' operators. One pair (using '#' and '##') removes a given pattern from the head of the string, returning the rest, while the other pair (using '%' and '%%') removes a pattern from the tail of the string. In each case, the form with one symbol removes the shortest matching pattern, while the one with two symbols removes the longest matching pattern. Two typical uses are:

```
% print $HOME
/home/pws
% print ${HOME##*/}
pws
% print ${HOME%/*}
/home
```

which here have the same effect of `${HOME:t}` and and `${HOME:h}`, and in zsh you would be more likely to use the latter. However, as you can see the pattern forms are much more general. Note the difference from:

```
% print ${HOME#*/}
home/pws
% print ${HOME%%/*}
```

where the shortest match of '*/' at the head was just the first slash, since '*' can match an empty string, while the longest match of '/*' at the tail was the entire string, right back to the first slash. Although these are standard forms, remember that the full power of zsh patterns is available.

How do you remember which operator does what? The fact that the longer form does the longer match is probably easy. Remembering that '#' removes at the head and '%' at the tail is harder. Try to think of 'hash' and 'head' (if you call it a 'pound sign', when it's nothing of the sort since a pound since looks like '£', you will get no sympathy from me), and 'percent' and 'posterior'. It never worked for me, but maybe I just don't have the mental discipline. Oliver Kiddle points out that '#' is further to the left (head) on a standard US keyboard. On my UK keyboard, '#' is right next to the return key, unfortunately, although here the confusion with 'pound sign' will jog your memory.

The most important thing to remember is: this notation is not our fault. Sorry, anyway. By the way, notice there's no funny business with colons in the case of the pattern operators. (Well — except for the zsh variant noted below.)

Zsh-specific parameter substitutions

Now for some enhancements that zsh has for using the forms of parameter substitution I've just given as well as some similar but different ones.

One simple enhancement is that in addition to '${*param*=*value*}' and '${*param*:=*value*}', zsh has '${*param*::=*value*}' which performs an unconditional assignment as well as sticking the value on the command line. It's not really any different from using a normal assignment, then a normal parameter substitution, except that zsh users like densely packed code.

All the assignment types are affected by the parameter flags 'A' and 'AA' which tell the shell to perform array and associative array assignment (in the second case, you need pairs of key/value elements as usual). You need to be a little bit careful with array elements and word splitting, however:

```
% print -l ${(A)foo::=one two three four}
one two three four
% print ${#foo}
1
```

That made $foo an array all right, but treated the argument as a scalar value and assigned it to the first element. There's a way round this:

```
% print -l ${(A)=foo::=one two three four}
one
two
three
four
% print ${#foo}
4
```

Here, the '=' *before* the parameter name has a completely different effect from the others: it turns on word-splitting, just as if the option SH_WORD_SPLIT is in effect. You may remember I went into this in appalling detail in the section 'Function parameters' in chapter 3.

You should be careful, however, as more sophisticated attempts at putting arrays inside parameter values can easily lead you astray. It's usually much easier to use the '*array*=(...)' or 'set -A ...' notations.

One extremely useful zsh enhancement is the notation '${+foo}' which returns 1 if $foo is set and 0 if it isn't. You can use this in arithmetic expressions. This is a much more flexible way of dealing with possibly unset parameters than the more standard '${foo?goodbye}' notation, and consequently is better used by zsh programmers. The notation 'plus foo' for 'foo is set' should be fairly memorable, too. A more standard way of doing this (noted by David Korn) is '0${foo+1}', giving 0 if $foo is not set and 01 if it is.

Parameter flags and pattern substitutions

Zsh increases the usefulness of the 'top and tail' operators with some of its parameter flags. Usually these show you what's left after the removal of some matched portion. However, with the flag (M) the shell will instead show you the matched portion itself. The flag (R) is the opposite and shows the rest: that's not all that useful in the normal case, since you get that by default. It only starts being useful when you combine it with other flags.

Next, zsh allows you to match on substrings, not just on the head or tail. You can do this by giving the flag (S) with either of the '#' or '%' pattern-matching forms. The difference here is whether the shell starts searching for a matching substring at the start or end of the full string. Let's take

```
foo='where I was huge lizards walked here and there'
```

and see what we get matching on 'h*e':

```
% print -l ${(S)foo#h*e} ${(S)foo##h*e} ${(S)foo%h*e} ${(S)foo%%h*e}
wre I was huge lizards walked here and there
w
where I was huge lizards walked here and tre
where I was huge lizards walked here and t
```

There are some odd discrepancies at first sight, but here's what happens. In the first case, '#' the shell looks forward until it finds a match for 'h*e', and takes the shortest, which is the 'he' in the first word. With '##', the match succeeds at the same point, but the longest match extends to the 'e' right at the end of the string. With the other two forms, the shell starts scanning backwards from the end, and stops as soon as it reaches a starting point which has a match. For both '%' and '%%' this is the last 'h', but the former matches 'he' and the latter matches 'here'.

You can extend this by using the (I) flag to specify a numeric index. The index needs to be delimited, conventionally, although not necessarily, by colons. The shell will then scan forward or backward, depending on the form used, until it has found the (I)'th match. Note that it only ever counts a single match from each position, either the longest or the shortest, so the (I)'th match starts from the (I)'th position which has any match. Here's what happens when we remove all the matches for '#' using the example above.

```
% for (( i = 1; i <= 5; i++ )); do
for> print ${(SI:$i:)foo#h*e}
for> done
wre I was huge lizards walked here and there
where I was  lizards walked here and there
where I was huge lizards walked re and there
where I was huge lizards walked here and tre
where I was huge lizards walked here and there
```

Each time we match and remove one of the possible 'h*e' sets where there is no 'e' in the middle, moving from left to right. The last time there was nothing left to match and the complete string was returned. Note that the index we used was itself a parameter.

It's obvious what happens with '##': it will find matches at all the same points, but they will all extend to the 'e' at the end of the string. It's probably less obvious what happens with '%%' and '%', but if you try it you will find they produce just the same set of matches as '##' and '#', respectively, but with the indices in the reverse order (4 for 1, 3 for 2, etc.).

You can use the 'M' flag to leave the matched portion rather than the rest of the string, if you like. There are three other flags which let you get the indices associated with the match instead of the string: (B) for the beginning, using the usual zsh convention where the first character is 1, (E) for the character *after* the end, and (N) for the length, simply B-E. You can even have more than one of these; the value substituted is a string with the given values with spaces between, always in the order beginning, end, length.

There is a sort of opposite to the '(S)' flag, which instead of matching substrings will only match the whole string; to do this, put a colon before the '#'. Hence:

```
% print ${foo:#w*g}
where I was huge lizards walked here and there
% print ${foo:#w*e}

%
```

The first one didn't match, because the '`g`' is not at the end; the second one did, because there is an '`e`' at the end.

Pattern replacement

The most powerful of the parameter pattern-matching forms has been borrowed from bash and ksh93; it doesn't occur in traditional Bourne shells. Here, you use a pair of '`/`'s to indicate a pattern to be replaced, and its replacement. Lets use the lizards again:

```
% print ${foo/h*e/urgh}
wurgh
```

A bit incomprehensible: that's because like most pattern matchers it takes the longest match unless told otherwise. In this case the `(S)` flag has been pressed into service to mean not a substring (that's automatic) but the shortest match:

```
% print ${(S)foo/h*e/urgh}
wurghre I was huge lizards walked here and there
```

That only replace the first match. This is where '`//`' comes in; it replaces every match:

```
% print ${(S)foo//h*e/urgh}
wurghre I was urgh lizards walked urghre and turghre
```

(No doubt you're starting to feel like a typical anachronistic Hollywood cave-dweller already.) Note the syntax: it's a little bit like substitution in `sed` or perl, but there's no slash at the end, and with '`//`' only the first slash is doubled. It's a bit confusing that with the other pattern expressions the single and double forms mean the shortest and longest match, while here it's the flag `(S)` that makes the difference.

The index flag `(I)` is useful here, too. In the case of '`/`', it tells the shell which single match to substitute, and in the case of '`//`' it tells the shell at which match to start: all matches starting from that are replaced.

Overlapping matches are never replaced by '`//`'; once it has put the new text in for a match, that section is not considered further and the text just to its right is examined for matches. This is probably familiar from other substitution schemes.

You may well be thinking 'wouldn't it be good to be able to use the matched text, or some part of it, in the replacment text?' This is what you can do in sed with '`\1`' or '`\&`' and in perl with '`$1`' and '`$&`'. It turns out this *is* possible with zsh, due to part of the more sophisticated pattern matching features. I'll talk about this when we come on to patterns, since it's not really part of parameter substitution, although it's designed to work well with that.

5.4.4 Flags for options: splitting and joining

There are three types of flag that don't look like flags, for historical reasons; you've already seen them in chapter 3. The first is the one that turns on the `SH_WORD_SPLIT` option, `${=foo}`. Note that you can mix this with flags that *do* look like flags, in parentheses, in which case the '`=`' must come after the closing parenthesis. You can force the option to

be turned off for a single substitution by doubling the symbol: '`${==foo}`'. However, you wouldn't do that unless the option was already set, in which case you are probably trying to be compatible with some other shell, and wouldn't want to use that form.

More control over splitting and joining is possible with three of the more standard type of flags, `(s)`, `(j)` and `(z)`. These do splitting on a given string, joining with a given string, and splitting just the way the shell does it, respectively. In the first two cases, you need to specify the string in the same way as you specified the index for the `(I)` flag. So, for example, here's how to turn `$PATH` into an ordinary array without using `$path`:

```
% print -l ${(s.:.)PATH}
/home/pws/bin
/usr/local/bin
/usr/sbin
/sbin
/bin
/usr/bin
/usr/X11R6/bin
/usr/games
```

Any character can follow the `(s)` or `(j)`; the string argument lasts until the matching character, here '.'. If the character is one of the bracket-like characters including '<', the 'matching' character is the corresponding right bracket, e.g. '`${(s<:>)PATH}`' and '`${(s(:))PATH}`' are both valid. This applies to all flags that need arguments, including `(I)`.

Although the split or join string isn't a pattern, it doesn't have to be a single character:

```
% foo=(array of words)
% print ${(j.**.)foo}
array**of**words
```

The `(z)` flag doesn't take an argument. As it handles splitting on the full shell definition of a word, it goes naturally with quoted expressions, and I discussed above its use with the `(Q)` flag for extracting words from a line with the quotes removed.

It's possible for the same parameter expression to have both splitting and joining applied to it. This always occurs in the same order, regardless of how you specify the flags: joining first, then splitting. This is described in the (rather hairy) complete set of rules in the manual entry for parameter substitution. There are one or two occasions where this can be a bit surprising. One is when you have `SH_WORD_SPLIT` set and try to join a string:

```
% setopt shwordsplit
% foo=('another array' of 'words with spaces')
% print -l ${(j.:.)foo}
another
array:of:words
with
spaces
```

You might not have noticed if you didn't use the '`-l` option to print, but the spaces still caused word-spliting even though you asked for the array to be joined with colons. To avoid this, either don't use `SH_WORD_SPLIT` (my personal preference), or use quotes:

```
% print -l "${(j.:.)foo}"
another array:of:words with spaces
```

The elements of an array would normally be joined by spaces in this case, but the character specified by the `(j)` flag takes precedence. In just the same way, if `SH_WORD_SPLIT` is in effect, any splitting string given by `(s)` is used instead of the normal set of characters, which are any characters that occur in the string `$IFS`, by default space, tab, newline and NUL.

Specifying a split for a particular parameter substitution not only sets the string to split on, but also ensures the split will take place even if the expression is quoted:

```
% array=('element one' 'element two' 'element three')
% print -l "${=array}"
element
one
element
two
element
three
```

To be clear about what's happening here: the quotes force the elements to be joined with spaces, giving a single string, which is then split on the original spaces as well as the one used to join the elements of the array.

I will talk shortly about nested parameter substitution; you should also note that splitting and joining will if necessary take place at all levels of a nested substitution, not just the outermost one:

```
% foo="three blind words"
% print ${#${(z)foo}}
3
```

This prints the length of the innermost expression; because of the zplit, that has produced a three-element array.

5.4.5 Flags for options: `GLOB_SUBST` and `RC_EXPAND_PARAM`

The other two flags that don't use parentheses affect options for single substitutions, too. The second is the '~' flag that turns on `GLOB_SUBST`, making the result of a parameter substitution eligible for pattern matching. As the notation is supposed to indicate, it also makes filename expansion possible, so

```
% foo='~'
% print ${~foo}
/home/pws
```

It's that first '~' which is giving the home directory; the one in the parameter expansion simply allows that to happen. If you have `GLOB_SUBST` set, you can use '`${~~foo}`' to turn it off for one substitution.

There's one other of these option flags: '^' forces on `RC_EXPAND_PARAM` for the current substitution, and '^^' forces it off. In chapter 3, I showed how parameters expanded with this option on fitted in with brace expansions.

5.4.6 Yet more parameter flags

Here are a few other parameter flags; I'm repeating some of these. A very useful one is '`t`' to tell you the type of a parameter. This came up in chapter 3 as well. It's most common use is to test the basic type of the parameter before trying to use it:

```
if [[ ${(t)myparam} != *assoc* ]]; then
  # $myparam is not an associative array.  Do something about it.
fi
```

Another very useful type is for left or right padding of a string, to a specified length, and optionally with a specified fill string to use instead of space; you can even specify a one-off string to go right next to the string in question.

```
foo='abcdefghij'
for (( i = 1; i <= 10; i++ )); do
 goo=${foo[1,$i]}
 print ${(l:10::X::Y:)goo} ${(r:10::X::Y:)goo}
done
```

prints out the rather pretty:

```
XXXXXXXXYa aYXXXXXXXX
XXXXXXXYab abYXXXXXXX
XXXXXXYabc abcYXXXXXX
XXXXXYabcd abcdYXXXXX
XXXXYabcde abcdeYXXXX
XXXYabcdef abcdefYXXX
XXYabcdefg abcdefgYXX
XYabcdefgh abcdefghYX
Yabcdefghi abcdefghiY
abcdefghij abcdefghij
```

Note that those colons (which can be other characters, as I explained for the `(s)` and `(j)` flags) always occur in pairs before and after the argument, so that with three arguments, the colons in between are doubled. You can miss out the '`:Y:`' part and the '`:X:`' part and see what happens. The fill strings don't need to be single characters; if they don't fit an exact number of times into the filler space, the last repetition will be truncated on the end furthest from the parameter argument being inserted.

Two parameters tell the shell that you want something special done with the value of the parameter substitution. The `(P)` flag forces the value to be treated as a parameter name, so that you get the effect of a double substitution:

```
% final=string
% intermediate=final
% print ${(P)intermediate}
string
```

This is a bit as if `$intermediate` were what in ksh is called a 'nameref', a parameter that is marked as a reference to another parameter. Zsh may eventually have those, too; there are places where they are a good deal more convenient than the '`(P)`' flag.

A more powerful flag is `(e)`, which forces the value to be rescanned for all forms of single-word substitution. For example,

```
% foo='$(print $ZSH_VERSION)'
% print ${(e)foo}
4.0.2
```

made the value of `$foo` be re-examined, at which point the command substitution was found and executed.

The remaining flags are a few simple special formatting tricks: order array elements in normal lexical (character) order with `(o)`, order in reverse order with `(O)`, do the same case-independently with `(oi)` or `(Oi)` respectively, expand prompt '`%`'-escapes with `(%)` (easy to remember), expand backslash escapes as `print` does with `p`, force all characters to uppercase with `(U)` or lowercase with `(L)`, capitalise the first character of the string or each array element with `(C)`, show up special characters as escape sequences with `(V)`. That should be enough to be getting on with.

5.4.7 A couple of parameter substitution tricks

I can't resist describing a couple of extras.

Zsh can do so much on parameter expressions that sometimes it's useful even without a parameter! For example, here's how to get the length of a fixed string without needing to put it into a parameter:

```
% print ${#:-abcdefghijklm}
13
```

If the parameter whose name you haven't given has a zero length (it does, because there isn't one), use the string after the '`:-`' instead, and take it's length. Note you need the colon, else you are asking the shell to test whether a parameter is set, and it becomes rather upset when it realises there isn't one to test. Other shells are unlikely to tolerate any such syntactic outrages at all; the `#` in that case is likely to be treated as `$#`, the number of shell arguments. But zsh knows that's not going to have zero length, and assumes you know what you're doing with the extra part; this is useful, but technically a violation of the rules.

Sometimes you don't need anything more than the flags. The most useful case is making the 'fill' flags generate repeated words, with the effect of perl's '`x`' operator (for those not familiar with perl, the expression '`"string" x 3`' produces the string 'stringstringstring'. Here, you need to remember that the fill width you specify is the total width, not the number of repetitions, so you need to multiply it by the length of the string:

```
% print ${(l.18..string.)}
stringstringstring
```

5.4.8 Nested parameter substitutions

Zsh has a system for multiple nested parameter substitutions. Whereas in most shells or other scripting languages you would do something like:

```
% p=/directory/file.ext
% p2=${p##*/}            # remove longest match of */ from head
% print $p2
file.ext
% print ${p%.*}          # remove shortest match of .* from tail
file
```

in zsh you can do this in one substitution:

```
% p=/directory/file.ext
% print ${${p##*/}%.*}
file
```

saving the temporary parameter in the middle. (Again, you are more likely to use `${p:t:r}` in this particular case.) Where this becomes a major advantage is with arrays: if `$p` is an array, all the substitutions are applied to every element of the array:

```
% p=(/dir1/file1.ext1 /dir2/file2.ext2)
% print ${${p##*/}%.*}
file1 file2
```

This can result in some considerable reductions in the code for processing arrays. It's a way of getting round the fact that an ordinary command line interface like zsh, designed originally for direct interaction with the user, doesn't have all the sophistication of a non-interactive language like perl, whose '`map`' function would probably be the neatest way of doing the same thing:

```
# Perl code.
@p = qw(/dir1/file1.ext1 /dir2/file2.ext2);
@q = map { m%^(?:.*/)(.*?)(?:\.[^.]*|)$%; } @p;
print "@q\n";'
```

or numerous possible variants. In a shell, there's no way of putting functions like that into the command line without complicating the basic 'command, arguments' syntax; so we resort to trickery with substitutions. Note, however, that this degree of brevity makes for a certain lack of readability even in Perl. Furthermore, zsh is so optimised for common cases that

```
print ${p:t:r}
```

will work for both arrays and scalars: the `:t` takes only the tail of the filename, stripping the directories, and the `:r` removes the suffix. These two operators could have slightly unexpected effects in versions of zsh before 4.0.1, removing 'suffixes' which contained directory paths, for example (though this is what the pattern forms taken separately do, too).

Note one feature of the nested substitution: you might have expected the '`${...}`' inside the other one to do a full parameter substitution, so that the outer one would act on the value of that — that's what you'd get if the substitution was on its own, after all. However, that's not what happens: the '`${...}`' inside is simply a syntactic trick to say 'here come more operations on the parameter'. This means that

```
bar='this doesn'\''t get substituted'
foo='bar'
print ${${foo}}
```

simply prints 'bar', not the value of $bar. This is the same case we had before but without any of the extra '##' and '%' bits. The reason is historical: when the extremely useful nested substitution feature was added, it was much simpler to have the leading '$' indicate to the shell that it should call the substitution function again than find another syntax. You can make the value be re-interpreted as another parameter substitution, using the (P) substitution flag described above. Just remember that ${${foo}} and ${(P)foo} are different.

5.5 That substitution again

Finally, here is a brief explanation of how to read the expression at the top of the chapter. This is for advanced students only (nutcases, if you ask me). You can find all the bits in the manual, if you try hard enough, even the ones I didn't get around to explaining above. As an example, let's suppose the array contains

```
array=(long longer longest short brief)
```

and see what

```
print ${array[(r)${(l.${#${(O@)array//?/X}[1]}..?.)}]}
```

gives.

1. Always start from the inside. The innermost expression here is

   ```
   ${(O@)array//?/X}
   ```

   Not much clearer? Start from the inside again: there's the parameter we're operating on, whose name is array. Before that there are two flags in parenthesis: (O) says sort the result in descending alphabetic order, (@) treat the result as an array, which is necessary because this inner substitution occurs where a scalar value (actually, an arithmetic expression) would usually occur, and we need to take an array element. After the array name, '//?/X' is a global substitution: take the pattern '?' (any character) wherever it occurs, and replace it with the string 'X'. The result of this is an array like $array, but with all the elements turned into strings consisting of 'X's in place of the original characters, and with the longest first, because that's how reverse alphabetic order works for strings with the same character. So

   ```
   long longer longest short brief
   ```

   would have become

   ```
   XXXXXXX XXXXXX XXXXX XXXXX XXXX
   ```

2. Next, we have '`${#`*result*`[1]}`' wrapped around that. That means that we take the first element of the array we arrived at above (the '`[1]`': that's why we had to make sure it was treated as an array), and then take the length of that (the '`#`'). We will end up in this case with 7, the length of the first (and longest element). We're finally getting somewhere.

3. The next step is the '`${(l.`*result*`..?.)}`'. Our previous *result* appears as an argument to the '`(l)`' flag of the substitution. That's a rather special case of nested substitution: at this point, the shell expects an arithmetical expression, giving the minimum length of a string to be filled on the left. The previous substitution was evaluated because arithmetic expressions undergo parameter substitution. So it is the result of that, 7, which appears here, giving the more manageable

   ```
   ${(l.7..?.)}
   ```

   The expression for the '`(l)`' flag in full says 'fill the result of this parameter substitution to a minimum width of 7 using the fill character '`?`'. What is the substitution we are filling? It's empty: zsh is smart enough to assume you know what you're doing when you don't give a parameter name, and just puts in an empty string instead. So the empty string is filled out to length 7 with question marks, giving '`???????`'.

4. Now we have '`${array[(r)???????]}`'. It may not be obvious (congratulations if the rest is), but the question marks are active as a pattern. Subscripts are treated specially in this respect. The subscript flag '`(r)`' means 'reverse match', not reverse as in backwards, but as in the opposite way round: search the array itself for a matching value, rather than taking this as an index. The only thing that will match this is a string of length 7. Bingo! that must be the element 'longest' in this case. If there were other elements of the same length, you would only get the first of that length; I haven't thought of a way of getting all the elements of that length substituted by a single expression without turning `$array` into an associative array, so if you have, you should feel smug.

After I wrote this, Sven Wischnowsky (who is responsible for a large fraction of the similar hieroglyphics in the completion functions) pointed out that a similar way of achieving this is:

```
print ${(M)array:#${~${(O@)array//?/?}[1]}}
```

which does indeed show all the elements of the maximum length. A brief summary of how this works is that the innermost expression produces an array of '`?`' corresponding to the elements, longest first in the way we did above, turning the '`?`' into pattern match characters. The next expansion picks the longest. Finally, the outermost expansion goes through `$array` to find elements which match the complete string of '`?`' and selects out those that do match.

If you are wondering about how to do that in perl in a single expression, probably sorting on length is the easiest:

```
# Perl code
@array = qw(long longer longest short brief);
@array = sort { length $b <=> length $a } @array;
```

and taking out the first element or first few elements of `@array`. However, in a highly-optimized scripting language you would almost certainly do it some other way: for example, avoid sorting and just remember the longest element:

```
# Perl code
$elt = '';
$l = 0;
foreach (@array) {
  $newl = length $_;
  $elt = $_, $l = $newl  if $l > $newl;
}
print $elt, "\n";
```

You can do just the same thing in zsh easily enough in this case;

```
local val elt
integer l newl
for val in $array; do
  newl=${#val}
  if (( newl > l )); then
    elt=$val
    (( l = newl ))
  fi
done
print $elt
```

so this probably isn't a particularly good use for nested substitution, even though it illustrates its power.

If you enjoyed that expression, there are many more like it in the completion function suite for you to goggle at.

5.6 Arithmetic Expansion

Performing mathematics within the shell was first described in chapter 3 where I showed how to create numeric parameters with variants of '`typeset`', and said a little about arithmetic substitution.

In addition to the math library, loadable with '`zmodload zsh/mathfunc`', zsh has essentially all the operators you expect from C and other languages derived from it. In other words, things like

```
(( foo = bar ? 3 : 1, ++brr ))
```

are accepted. The comma operator works just as in C; all the arguments are evaluated, in this case '`foo = bar ? 3 : 1`' assigns 3 or 1 to `$foo` depending whether or not `bar` is non-zero, and then `$brr` is incremented by 1. The return status is determined by the final expression, so if `$brr` is zero after increment the return status is one, else it is zero (integers may be negative).

One extra operator has been borrowed from FORTRAN, or maybe Perl, the exponentiation operator, '`**`'. This can take either integers or floating point numbers, though a negative exponent will cause a floating point number to be returned, so '`$(( 2 ** -1 ))`' gives you 0.5, not rounded down to zero. This is why the standard library function `pow` is missing from `zsh/mathfunc` — it's already there in that other form. Pure integer exponentiation,

however, is done by repeated multiplication — up to arbitrary sizes, so instead of '`2 ** 100`', you should use '`1 « 100`', and for powers of any other integer where you don't need an exact result, you should use floating point numbers. For this purpose, the `zsh/mathfunc` library makes 'casts' available; '`float`(*num*)' forces the expression *num* to interpreted as a floating point number, whatever it would otherwise have given, although the trick of adding '`0.0`' to a number works as well. Note that, although this works like a cast in C, the syntax is that of an ordinary function call. Likewise, '`int`(*num*)' causes the number to be interpreted as an integer — rounding towards zero; you can use `floor` and `ceil` to round down or up, and `rint` to round to the nearest integer, although these three actually produce floating point numbers. They are standard C library functions.

For completeness, the assignment form of exponentiation '`**=`' also works. I can't remember ever using it.

The range of integers depends on how zsh was configured on your machine. The primary goal is to make sure integers are large enough to represent indexes into files; on some systems where the hardware usually deals with 32-bit integers, file sizes may be given by 64-bit integers, and zsh will try to use 64-bit integers as well. However, zsh will test for large integers even if no large file support is available; usually it just requires that your compiler has some easy to recognise way of defining 64-bit integers, such as '`long long`' which may be handled by gcc even if it isn't by the native compiler. You can easily test; if your zsh supports 64-bit integers, the largest available integer is:

```
% print $(( 0x7FFFFFFFFFFFFFFF ))
9223372036854775807
```

and if you try adding something positive to that, you will get a negative result due to two's complement arithmetic. This should be large enough to count most things.

The range of floating point numbers is always that of a C '`double`', which is usually also 64 bits, and internally the number is highly likely to be in the IEEE standard form, which also affects the precision and range you can get, though that's system specific, too. On most systems, the math library functions handle `double`s rather than single precision `float`s, so this is the natural choice. The cast function is called '`float`' because, unlike C, the representation of a floating point number is chosen for you, so the generic name is used.

5.6.1 Entering and outputting bases

I'll say a word or two about bases. I already said you could enter a number with any small base in a form like '`2#101010`' or '`16#ffff`', and that the latter could also be '`0xffff`' as in C. You can't, however, enter octal numbers just by using a leading '`0`', which you might expect from C. Here's an example of why not. Let's set:

```
% foo=${(%):-%D}
% print $foo
01-08-06
```

The first line is another of those bogus parameter substitutions where we gave it a literal string and a blank parameter. We also gave it the flag '`(%)`', which forces prompt escapes to be expanded, and in prompts '`(%D)`' is the date as *yy-mm-dd*. Let's write a short program to find out what the date after `$foo` is. We have the luxury of 99 years to worry about the century wrapping, so we'll ignore it (and the Gregorian calendar).

```
mlens=(31 28 31 30 31 30 31 31 30 31 30 31)
date=(${(s.-.)foo})    #  splits to array (01 08 23)
typeset -Z 2 incr
if (( ${date[3]} < ${mlens[${date[2]}]} )); then
  # just increment day
  (( incr = ${date[3]} + 1 ))
  date[3]=$incr
else
  # go to first of next month
  date[3]=01
  if (( ${date[2]} < 12 )); then
    (( incr = ${date[2]} + 1 ))
    date[2]=$incr
  else
    # happy new year
    date[2]=01
    (( incr = ${date[3]} + 1 ))
    date[3]=$incr
  fi
fi
print ${date[1]}-${date[2]}-${date[3]}
```

This will print '`01-08-07`'. Before I get to the point, various other explanations. We forced `$foo` to be split on any '–' in it, giving a three-part array. The next trick was '`typeset -Z 2 incr`', which tells the shell that `$incr` is to be at least two characters, filled with leading zeroes. That's how we got the '`07`' at the end, instead of just '`7`'. There's another way of doing this: replace

```
typeset -Z 2 incr
(( incr = ${date[2]} + 1 ))
date[2]=$incr
```

with:

```
date[2]=${(l.2..0.)$(( ${date[2]} + 1 ))}
```

This uses the `(l)` parameter flag to fill up to two characters with a zero (the default is a space, so we need to specify the '`0`' this time), using the fact that parameter operations can have a nested `$`-substution. This second form is less standard, however.

Now, finally, the point. In that '$((${date[2]} + 1))', the '`${date[2]}`' is simply the *scalar* '`08`' — the result of splitting an arbitrary string into an array. Suppose we used leading zeroes to signify octal numbers. We would get something like:

```
% print $(( ${date[2]} + 1 ))
zsh: bad math expression: operator expected at `8 + 1 '
```

because the expression in the substitution becomes '`08 + 1`' and an 8 can't appear in an octal number. So we would have to strip off any otherwise harmless leading zeroes. Parsing dates, or indeed strings with leading zeroes as padding, is a fairly common thing for a shell to do, and octal arithmetic isn't. So by default leading zeroes don't have that effect.

However, there is an option you can set, OCTAL_ZEROES; this is required for compatibility with the POSIX standard. That's how I got the error message in the previous paragraph, in fact.

Floating point numbers are never octal, always decimal:

```
% setopt octalzeroes
% print $(( 077 ))
63
% print $(( 077.43 ))
77.430000000000007
```

The other option to do with bases is C_BASES, which makes hexadecimal (and, if you have OCTAL_ZEROES set, octal) numbers appear in the form that you would use as input to a C (or, once again, Perl) program.

How do you persuade the shell to print out numbers in a particular base anyway? There are two ways. The first is to associate a base with a parameter, which you do with an argument after the '-i' option to typeset:

```
% typeset -i 16 hexnum=32
% print $hexnum
16#20
```

This is the standard way. By the way, there's a slight catch with bases, taken over from ksh: if you *don't* specify a base, the first assignment will do the job for you.

```
% integer anynum
% (( anynum = 16#20 ))
% print $anynum
16#20
```

Only constants with explicit bases in an expression produce this effect; the first time 'anynum' comes into contact with a '*base*#*num*', or a hexadecimal or (where applicable) octal expression in the standard C form, it will acquire a default output base. So you need to use 'typeset -i 10' if you don't like that.

Often, however, you just want to print out an expression in, say, hexadecimal. Zsh has a shorthand for this, which is only in recent versions (and not in other shells). Preceding an expression by '[#*base*]' causes the default output base to be set to base with the the usual prefix showing the base, and '[##*base*]' will do the same but without the prefix, i.e. '$(([##16]255))' is simply 'FF'. This has no effect on assignments to a parameter, not even on the parameter's default output base, but it will affect the result of a direct substitution using $((...)).

5.6.2 Parameter typing

Just as creating a parameter with an ordinary assignment makes it a scalar, so creating it in an arithmetic substitution makes it either an integer or a floating point parameter, according to the value assigned. This is likely to be a floating point number if there was a floating point number in the expression on the right hand side, and an integer otherwise. However, there

are reasons why a floating point number on the right may not have this effect — use of `int`, for example, since it produces an integer.

However, relying on implicit typing in this fashion is bad. One of the reasons is explained in the manual entry, and I can't do better than use that example (since I wrote it):

```
for (( f = 0; f < 1; f += 0.1 )); do
  print $f
done
```

If you try this, and `$f` does not already exist, you will see an endless stream of zeroes. What's happening is that the original assignment creates `$f` as an integer to store the integer `0` in. After printing this, `$f` is incremented by adding `0.1` to it. But once created, `$f` remains an integer, so the resulting `0.1` is cast back to an integer, and the resulting zero is stored back in `$f`. The result is that `$f` is never incremented.

You could turn the first `0` into `0.0`, but a better way is to declare '`float f`' before the loop. In a function, this also ensures `$f` is local to the function.

If you use a scalar to store an integer or floating point, everything will work. You don't have the problem just described, since although `$f` contains what looks like an integer to start with, it has no numeric type associated with it, and when you store `0.1` into `$f`, it will happily overwrite the string '`0`'. It's a bit more inefficient to use scalars, but actually not that much. You can't specify an output base or precision, and in versions of zsh up to 4.0.x, there is a problem when the parameter already has a string in it which doesn't make sense as a numeric expression:

```
% foo='/file/name'
% (( foo = 3 ))
zsh: bad math expression: operand expected at `/file/name'
```

The unexpected error comes because '`/file/name/`' is evaluated even though the shell is about to overwrite the contents of `$foo`. Versions of the shell from 4.1.1 have a fix for this, and the integer assignment works as expected.

You need to be careful with scalars that might contain an empty string. If you declare '`integer i`', it will immediately contain the value 0, but if you declare '`typeset s`', the scalar `$s` will just contain the empty string. You get away with this if you use the parameter without a '`$`' in front:

```
% typeset s
% print $(( 3 * s ))
0
```

because the math code tries to retrieve `$s`, and when it fails puts a `0` there. However, if you explicitly use `$s`, the math code gets confused:

```
% print $(( 3 * $s ))
zsh: bad math expression: operand expected at `'
```

because '`$s`' evaluates to an empty string before the arithmetic evaluation proper, which spoils the syntax. There's one common case where you need to do that, and that's with positional parameters:

```
% fn() { print "Twice $1 is $(( 2 * $1 ))"; }
% fn 3
Twice 3 is 6
% fn
fn: bad math expression: operand expected at `'
```

Obviously turning the '$1' into '1' means something completely different. You can guard against this with default values:

```
% fn() { print "Twice ${1:=0} is $(( 2 * $1 ))"; }
% fn
Twice 0 is 0
```

This assigns a default value for $0 if one was not set. Since parameter expansion is performed in one go from left to right, the second reference to $1 will pick up that value.

Note that you need to do this even if it doesn't look like the number will be needed:

```
% fn() { print $(( ${1:-0} ? $1 : 3 )); }
% fn
fn: bad math expression: operand expected at `: 3 '
```

The expression before the '?' evaluates to zero if $1 is not present, and you expect the expression after the colon to be used in that case. But actually it's too late by then; the arithmetic expression parser has received '0 ? : 3', which doesn't make sense to it, hence the error. So you need to put in '${1:-0}' for the second $1, too — or ${1:-32}, or any other number, since it won't be evaluated if $1 is empty, it just needs to be parsed.

You should note that just as you can put numbers into scalar parameters without needing any special handling, you can also do all the usual string-related tricks on numeric parameters, since there is automatic conversion in the other direction, too:

```
% float foo
% zmodload -i zsh/mathfunc
% (( foo = 4 * atan(1.0) ))
% print $foo
3.141592654e+00
% print ${foo%%.*}${foo##*.[0-9]##}
3e+00
```

The argument -i to zmodload tells it not to complain if the math library is already loaded. This gives us access to atan. Remember, 'float' declares a parameter whose output includes an exponent — you can actually convert it to a fixed point format on the fly using 'typeset -F foo', which retains the value but alters the output type. The substitution uses some EXTENDED_GLOB chicanery: the final '[0-9]##' matches one or more occurrences of any decimal digit. So the head of the string value of $foo up to the last digit after the decimal point is removed, and the remainder appended to whatever appears before the decimal point.

Starting from 4.1.1, a calculator function called zcalc is bundled with the shell. You type a standard arithmetic expression and the shell evaluates the formula and prints it out. Lines already entered are prefixed by a number, and you can use the positional parameter corresponding to that number to retrieve that result for use in a new formula. The function uses vared to read the formulae, so the full shell editing mechanism is available. It will also read in zsh/mathfunc if that is present.

5.7 Brace Expansion and Arrays

Brace expansion, which you met in chapter 3, appears in all csh derivatives, in some versions of ksh, and in bash, so is fairly standard. However, there are some features and aspects of it which are only found in zsh, which I'll describe here.

A complication occurs when arrays are involved. Normally, unquoted arrays are put into a command line as if there is a break between arguments when there is a new element, so

```
% array=(three separate words)
% print -l before${array}after
beforethree
separate
wordsafter
```

unless the `RC_EXPAND_PARAM` option is set, which combines the before and after parts with *each* element, so you get:

```
% print -l before${^array}after
beforethreeafter
beforeseparateafter
beforewordsafter
```

— the '^' character turns on the option just for that expansion, as '=' does with `SH_WORD_SPLIT`. If you think of the character as a correction to a proof, meaning 'insert a new word between the others here', it might help you remember (this was suggested by Bart Schaefer).

These two ways of expanding arrays interact differently with braces; the more useful version here is when the `RC_EXPAND_PARAM` option is on. Here the array acts as sort of additional nesting:

```
% array=(two three)
% print X{one,${^array}}Y
XoneY XtwoY XoneY XthreeY
```

with the `XoneY` tacked on each time, but because of the braces it appears as a separate word, so there are four altogether.

If `RC_EXPAND_PARAM` is not set, you get something at first sight slightly odd:

```
% array=(two three)
% print X{one,$array}Y
X{one,two three}Y
```

What has happened here is that the `$array` has produced two words; the first has '`X{one,`' tacked in front of the array's '`two`', while the second likewise has '`}Y`' on the end of the array's '`three`'. So by the time the shell comes to think about brace expansion, the braces are in different words and don't do anything useful.

There's no obvious simple way of forcing the `$array` to be embedded in the braces at the same level, instead of like an additional set of braces. There are more complicated ways, of course.

```
% array=(two three)
% print X${^=:-one $array}Y
XoneY XtwoY XthreeY
```

Yuk. We gave parameter substitution a string of words, the array with `one` stuck in front, and told it to split them on spaces (this will split on any extra spaces in elements of `$array`, unfortunately), while setting `RC_EXPAND_PARAM`. The parameter flags are '`^=`'; the '`:-`' is the usual 'insert the following if the substitution has zero length' operator. It's probably better just to create your own temporary array and apply `RX_EXPAND_PARAM` to that. By the way, if you had `RC_EXPAND_PARAM` set already, the last result would have been different becuase the embedded `$array` would have been expanded together with the '`one `' in front of it.

Braces allow numeric expressions; this works a little like in Perl:

```
% print {1..10}a
1a 2a 3a 4a 5a 6a 7a 8a 9a 10a
```

and you can ask the numbers to be padded with zeroes:

```
% print {01..10}b
01b 02b 03b 04b 05b 06b 07b 08b 09b 10b
```

or have them in descending order:

```
% print {10..1}c
10c 9c 8c 7c 6c 5c 4c 3c 2c 1c
```

Nesting this within other braces works in the expected way, but you can't have any extra braces inside: the syntax is fixed to number, two dots, number, and the numbers must be positive.

There's also an option `BRACE_CCL` which, if the braces aren't in either of the above forms, expands single letters and ranges of letters:

```
% setopt braceccl
% print 1{abw-z}2
1a2 1b2 1w2 1x2 1y2 1z2
```

An important point to be made about braces is that they are *not* part of filename generation; they have nothing to do with pattern matching at all. The shell blindly generates all the arguments you specify. If you want to generate only some arguments, depending on what files are matched, you should use the alternative-match syntax. Compare:

```
% ls
file1
% print file(1|2)
file1
% print file{1,2}
file1 file2
```

The first matches any of '`file1`' or '`file2`' it happens to find in the directory (regardless of other files). The second doesn't look at files in the directory at all; it simply expands the braces according to the rules given above.

This point is particularly worthy of note if you have come from a C-shell world, or use the CSH_NULL_GLOB option:

```
csh% echo file{1,2}
file1 file2
csh% echo f*{1,2}
file1
```

('csh%' is the prompt, to remind you if you're skipping through without reading the text), where the difference occurs because in the first case there was no pattern, so brace expansion was done on ordinary words, while in the second case the '*' made pattern expansion happen. In zsh, the sequence would be: 'f*{1,2}' becomes 'f*1 f*2'; the first becomes file1 and the second fails to match. With CSH_NULL_GLOB set, the failed match is simply removed; there is no error because one pattern has succeeded in matching. This is presumably the logic usually followed by the C shell. If you stick with 'file(1|2)' and 'f*(1|2)' — in this case you can simplify them to 'file[12]' and 'f*[12]', but that's not true if you have more than one character in either branch — you are protected from this difference.

5.8 Filename Expansion

Filename expansions consists of just '~/...', '~user/...', '~namedir/...' and '=prog', where the '~' and '=' must be the first character of a word, and the option EQUALS must be set (it is by default) for the '=' to be special. I told you about all this in chapter 3.

There's really only one thing to add, and that's the behaviour of the MAGIC_EQUAL_SUBST option. Assignments after typeset and similar statements are handled as follows

```
% typeset foo=~pws
% print $foo
/home/pws
% typeset PATH=$PATH:~pws/bin
% print ${path[-1]}
/home/pws/bin
```

It may not be obvious why this is not obvious. The point is that 'typeset' is an ordinary command which happens to be a shell builtin; the arguments of ordinary commands are not assignments. However, a special case is made here for typeset and its friends so that this works, even though, as I've said repeatedly, array assignments can't be done after typeset. The parameter $PATH isn't handled differently from any other — any colon in an assignment to any variable is special in the way shown.

It's often useful to have this feature with commands of your own. There is an option, MAGIC_EQUAL_SUBST, which spots the forms '...=~...' and '...=...:~...' for any command at all and expands ~-expressions. Commands where this is particularly useful include make and the GNU configure command used for setting up the compilation of a software package from scratch.

A related new option appeared in version 4.0.2 when it became clear there was an annoying difference between zsh and other shells such as ksh and bash. Consider:

```
export FOO=`echo hello there`
```

In ksh and bash, this exports `$foo` with the value '`hello there`'. In zsh, however, an unquoted backquote expression forces wordsplitting, so the line becomes

```
export FOO=hello there
```

and exports `$FOO` with the value '`hello`', and `$there` with any value it happens to have already or none if it didn't exist. This is actually perfectly logical according to the rules, but you can set the option `KSH_TYPESET` to have the other interpretation.

Normally, `KSH_TYPESET` applies only after parameter declaration builtins, and then only in the values of an assignment. However, in combination with `MAGIC_EQUAL_SUBST`, you will get the same behaviour with any command argument that looks like an assignment — actually, anything following an '=' which wasn't at the start of the word, so '`"hello mother, => I'm home "$(echo right now)`' qualifies.

It seems that bash behaves as if both `KSH_TYPESET` *and* `MAGIC_EQUAL_SUBST` are always in effect.

5.9 Filename Generation and Pattern Matching

The final topic is perhaps the biggest, even richer than parameter expansion. I'm finally going to explain the wonderful world of zsh pattern matching. In addition to patterns as such, you will learn such things as how to find all files in all subdirectories, searching recursively, which have a given name, case insensitive, are at least 50 KB large, no more than a week old and owned by the root user, and allowing up to a single error in the spelling of the name. In fact, the required expression looks like this:

```
**/(#ia1)name(LK+50mw-1u0)
```

which might appear, at first sight, a mite impenetrable. We'll work up to it gradually.

To repeat: filename generation is just the same as globbing, only longer. I use the terms interchangeably.

5.9.1 Comparing patterns and regular expressions

It can be confusing that there are two rather different sorts of pattern around, those used for matching files on a command line as in zsh and other shells, and those used for matching text inside files as in `grep`, `sed`, `emacs`, `perl` and many other utilities, each of which, typically, has a slightly different form for patterns (called in this case 'regular expressions', because UNIX was designed by computer scientists). There are even some utilities like TCL which provide both forms.

Zsh deals exclusively with the shell form, which I've been calling by its colloquial name, 'globbing', and consequently I won't talk about regular expressions in any detail. Here are the two classic differences to note. First, in a shell, '`*`' on its own matches any set of characters, while in a regular expression it always refers to the previous pattern, and says that that can be repeated any number of times. Second, in a shell '`.`' is an ordinary (and much used) character, while in a regular expression it means 'any character', which is specified by '`?`' in the shell. Put this together, and what a shell calls '`*`' is given by '`.*`' in a regular expression.

'*' in the latter case is called a 'Kleene closure': it's those computer scientists again. In zsh, art rather than science tends to be in evidence.

In fact, zsh does have many of the features available in regular expressions, as well as some which aren't. Remember that anywhere in zsh where you need a pattern, it's of the same form, whether it's matching files on the command line or a string in a `case` statement. There are a few features which only fit well into one or another use of patterns; for example the feature that selects files by examining their type, owner, age, etc. (the final parenthesis in the expression I showed above) are no use in matching against a string.

5.9.2 Standard features

There is one thing to note about the simple pattern matching features '*' and '?', which is that when matching file names (not in other places patterns are used, however) they never match a leading '.'. This is a convention in UNIX-like systems to hide certain files which are not interesting to most users. You may have got the impression that files begining with '.' are somehow special, but that's not so; only the files '.' (the current directory) and '..' (the parent directory, or the current directory in `/`) are special to the system. Other files beginning with '.' only appear special because of a conspiracy between the shell (the rule I've just given) and the command `ls`, which, when it lists a directory, doesn't show files beginning '.' unless you give the '`-a`' option. Otherwise '.'-files are perfectly normal files.

You can suppress the special rule for an initial '.' by setting the option `GLOB_DOTS`, in which case '*' will match every single file and directory except for '.' and '..'.

In addition to '*' and '?', which are so basic that even DOS had them (though I never *quite* worked out exactly what it was doing with them a lot of the time), the pattern consisting of a set of characters in square brackets appears in all shells. This feature happens to be pretty much the same as in regular expressions. '`[abc]`' matches any one of those three characters; '`[a-z]`' matches any character between `a` and `z`, inclusive; '`[^a-z]`' matches any single character *except* those 26 — but notice it still matches a single character.

A recent common enhancement to character ranges features in zsh, which is to specify types of characters instead of listing them; I'm just repeating the manual entry here, which you should consult for more detail. The special syntax is like '`[:`*spec*`:]`', where the square brackets there are in addition to the ones specifying the range. If you are familiar with the 'ctype' macros use in C programmes, you will probably recognise the things that *spec* can be: `alnum`, `alpha`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, `xdigit`. The similarity to C macros isn't just for show: the shell really does call the macro (or function) '`isalpha`' to test for `[:alpha:]`ness, and so on. On most modern systems which support internationalization this means the shell can tell you whether a character is, say, an alphabetic letter in the character set in use on your machine. By the way, zsh doesn't use international character set support for sorting matches — this turned out to produce too many unexpected effects.

So '`[^[:digit:]]`' matches any single character other than a decimal digit. Standards say you should use '`!`' instead of '^' to signify negation, but most people I know don't; also, this can clash with history substitution. However, it is accepted by zsh anywhere where history substitution doesn't get its hands on the '`!`' first (which includes all scripts and autoloaded functions).

5.9.3 Extensions usually available

Now we reach the bits specific to zsh. I've divided these into two parts, since some require the option 'EXTENDED_GLOB' to be set — those which are most likely to clash with other uses of the characters in question.

Numeric ranges

One possibility that is always available is the syntax for numeric ranges in the form '<*num1*-*num2*>'. You can omit either *num1*, which defaults to zero, or *num2*, which defaults to infinity, or both, in which case any set of digits will be matched. Note that this really *does* mean infinity, despite the finite range of integers; missing out *num2* is treated as a special case and the shell will simply advance over any number of digits. (In *very* old versions of zsh you had to use '<>' to get that effect, but that has been removed and '<>' is now a redirection operator, as in other shells; '<->' is what you need for any set of digits.)

I repeat another warning from the manual: this test

```
[[ 342 = <1-30>* ]]
```

succeeds, even though the number isn't in the range 1 to 30. That's because '<1-30>' matches '3' and '*' matches 42. There's no use moaning, it's a consequence of the usual rule for patterns of all types in shells or utilities: pattern operators are tried independently, and each 'uses up' the longest piece of the string it is matching without causing the rest of the match to fail. We would have to break this simple and well-tried rule to stop numeric ranges matching if there is another digit left. You can test for that yourself, of course:

```
[[ 342 = <1-30>(|[^[:digit:]]*) ]]
```

fails. I wrote it so that it would match any number between 1 and 30, either not followed by anything, or followed by something which doesn't start with a digit; I will explain what the parentheses and the vertical bar are doing in the next section. By the way, leading zeroes are correctly handled (and never force octal interpretation); so '00000003NaN' would successfully match the pattern.

The numbers in the range are always positive integers; you need extra pattern trickery to match floating point. Here's one attempt, which uses EXTENDED_GLOB operators, so come back and look when you've read the rest of this section if it doesn't make sense now:

```
isfloat() {
  setopt localoptions extendedglob
  if [[ $1 = ([-+]|)([0-9]##.[0-9]#|[0-9]#.[0-9]##)\
([eE]([-+]|)[0-9]##|) ]]; then
    print -r -- "$1 is a floating point number"
  else
    print -r -- "$1 is not a floating point number"
  fi
}
```

I've split it over two lines to fit. The first parenthesis matches an optional minus or plus sign — careful with '-' in square brackets, since if it occurs in the middle it's taken as a range,

and if you want it to match itself, it has to be at the start or end. The second parenthesis contains an alternative because '`.`' isn't a floating point number (at least, not in my book, and not in zsh's, either), but both '`0.`' and '`.0`' *are* properly formed numbers. So we need at least one digit, either before or after the decimal point; the '`##`' means 'at least one occurrence of the previous expression', while the '`#`' means 'zero or more occurrences of the previous expression'. The expresion on the next line matches an exponent; here you need at least one digit, too. So '`3.14159E+00`' is successfully matched, and indeed you'll find that zsh's arithmetic operations handle it properly.

The range operator is the only special zsh operator that you can't turn off with an option. This is usually not a problem, but in principle a string like '`<3-10>`' is ambiguous, since in another shell it would be read as '`<3-10 >`', meaning 'take input from file `3-10`, and send output to the file formed by whatever comes after the expression'. It's very unlikely you will run across this in practice, however, since shell code writers nearly alwys put a space after the end of a file name for redirection if something else follows on the command line, and that's enough to differentiate it from a range operator.

Parentheses

Parentheses are quite natural in zsh if you've used extended regular expressions. They are usually available, and only turned off if you set the '`SH_GLOB`' option to ensure compatibility with shells that don't have it. The key part of the expression is the vertical bar, which specifies an alternative. It can occur as many times as necessary; '`(a|b|c|d|e|f|g|h|i|j|k|l|m)`' is a rather idiosyncratic way of writing '`[a-m]`'. If you don't include the vertical bar (we'll see reasons for not doing so later), and you are generating filenames, you should be careful that the expression doesn't occur at the end of the pattern, else it would be taken as a 'glob qualifier', as described below. The rather unsightly hack of putting '`(|)`' (match the empty string or the empty string — guess what this matches?) right at the end will get around that problem.

The vertical bar usually needs to be inside parentheses so that the shell doesn't take it as a pipe, but in some contexts where this won't happen, such as a case statement label, you can omit any parentheses that would completely surround the pattern. So in

```
case $foo in
  (bar|rod|pipe) print "foo represents a piece of metal"
            ;;
  (*) print "Are you trying to be different?"
      ;;
esac
```

the surrounding parentheses are the required syntax for `case`, rather than pattern parentheses — the same syntax works in other shells. Then '`bar|rod`' is an ordinary zsh expression matching either `bar` or `rod`, in a context where the '`|`' can't be mistaken for a pipe. In fact, this whole example works with `ksh` — but there the use of '`|`' is a special case, while in zsh it fits in with the standard pattern rules.

Indeed, ksh has slightly different ways of specifying patterns: to make the use of parentheses less ambiguous, it requires a character before the left parenthesis. The corresponding form for a simple alternative is '`@(this|that)`'. The '`@`' can also be a '`?`', for zero or one occurrences of what's in the parentheses; '`*`' for any number of repetitions, for example '`thisthisthatthis`'; or '`!`' for anything except what's in the parentheses. Zsh allows this syntax if you set the option `KSH_GLOB`. Note that this is independent of the option `SH_GLOB`;

if you set KSH_GLOB but not SH_GLOB, you can actually use both forms for pattern matching, with the ksh form taking precedence in the case of ambiguities. This is probably to be avoided. In ksh emulation, both options are set; this is the only sensible reason I know of for using these options at all. I'll show some comparisons in the next section.

An important thing to note is that when you are matching files, you can't put directory separators inside parentheses:

```
# Doesn't work!
print (foo/bar|bar/foo)/file.c
```

doesn't work. The reason is that it's simply too difficult to write; pattern matching would be bound in a highly intricate way with searching the directory hierarchy, with the end of a group sending you back up to try another bit of the pattern on a directory you'd already visited. It's probably not impossible, but the pattern code maintainer (me) isn't all that enthusiastic about it.

5.9.4 Extensions requiring EXTENDED_GLOB

Setting EXTENDED_GLOB makes three new types of operator available: those which excluded a particular pattern from matching; those which specify that a pattern may occur a repeated number of times; and a set of 'globbing flags', a little bit like parameter flags which I'll describe in a later section since they are really the icing on the cake.

Negative matches or exclusions

The simpler of the two exclusions uses '^' to introduce a pattern which must *not* be matched. So a trivial example (I will assume for much of the rest of the chapter that the option EXTENDED_GLOB is set) is:

```
[[ foo = ^foo ]]
[[ bar = ^foo ]]
```

The first test fails, the second succeeds. It's important to realise that that the pattern demands nothing else whatever about the relevant part of the test string other than it doesn't match the pattern that follows: it doesn't say what length the matched string should have, for example. So

```
[[ foo = *^foo ]]
```

actually *does* match: * swallows up the whole string, and the remaining empty string successfully fails to be 'foo'. Remember the mantra: each part of the pattern matches the longest possible substring that causes the remainder of the pattern not to fail (unless, of course, failure is unavoidable).

Note that the ^ applies to the whole pattern to its right, either to the end of the string, or to the end of the nearest enclosing parenthesis. Here's a couple more examples:

```
[[ foo = ^foo* ]]
```

Overall, this fails to match: the pattern '`foo*`' always matches the string on the left, so negating means it always fails.

```
[[ foo = (^foo)* ]]
```

This is similar to the last example but one. The expression in the parenthesis first matches against `foo`; this causes the overall match to fail because of the `^`, so it backs up one character and tries again. Now '`fo`' is successfully matched by `^foo` and the remaining '`o`' is matched by the `*`, so the overall match succeeds. When you know about backreferences, you will be able to confirm that, indeed, the expression in parentheses matches '`fo`'. This is a quite subtle point: it's easy to imagine that '`^foo`' says 'match any three letter string except the one I've given you', but actually there is no requirement that it match three letters, or indeed any.

In filename generation, the `^` has a lower precedence than a slash:

```
% print /*/tmp
/data/tmp /home/tmp /usr/tmp /var/tmp
% print /^usr/tmp
/data/tmp /home/tmp /var/tmp
```

successfully caused the first level of directories to match anything but '`usr`'. A typical use of this with files is '`^*.o`' to match everything in a directory except files which end with '`.o`'.

Note one point mentioned in the FAQ — probably indicating the reason that '`^`' is only available with `EXTENDED_GLOB` switched on. Some commands use an initial '`^`' to indicate a control character; in fact, zsh's `bindkey` builtin does this:

```
bindkey '^z' backward-delete-word
```

which attaches the given function to the keystroke `Ctrl-z`. You must remember to quote that keystroke expression, otherwise it would expand to a list of all files in the current directory not called '`z`', very likely all of them.

There's another reason this isn't available by default: in some versions of the Bourne shell, '`^`' was used for pipes since '`|`' was missing on some keyboards.

The other exclusion operator is closely related. '*pat1*`~`*pat2*' means 'anything that matches *pat1* as long as it doesn't also match *pat2*'. If *pat1* is `*`, you have the same effect as '`^`' — in fact, that's pretty much how '`^`' is currently implemented.

There's one significant difference between '`*~`*pat*' and '`^`*pat*': the `~` has a *lower* precedence than '`/`' when matching against filenames. What's more, the pattern on the right of the `~` is not treated as a filename at all; it's simply matched against any filename found on the left, to see if it should be rejected. This sounds like black magic, but it's actually quite useful, particularly in combination with the recursive globbing syntax:

```
print **/*~*/CVS(/)
```

matches any subdirectory of the current directory to any depth, except for directories called `CVS` — the '`*`' on the right of the '`~`' will match any character including '`/`'. The final '`(/)`' is a glob qualifier indicating that only directories are to be allowed to match — note that

it's a positive assertion, despite being after the '~'. Glob qualifiers do not feel the effect of preceding exclusion operators.

Note that in that example, any subdirectory of a directory called `CVS` would have matched successfully; you can see from the pattern that the expression after the '~' wouldn't weed it out. Slightly less obviously, the '`**/*`' matches files in the current directory, while the '`*/CVS`' never matches a '`CVS`' in the current directory, so that could appear. If you want to, you can fix that up like this:

```
print **/*~(*/|)CVS(/*|)(/)
```

again relying on the fact that '/'s are not special after the '~'. This will ruthlessly weed out any path with a directory component called `CVS`. An easier, but less instructive, way is

```
print ./**/*~*/CVS(/)
```

You can restrict the range of the tilde operator by putting it in parentheses, so '`/(*~usr)/tmp`' is equivalent to '`/^usr/tmp`'.

A '~' at the beginning is never treated as excluding what follows; as you already know, it has other uses. Also, a '~' at the end of a pattern isn't special either; this is lucky, because Emacs produces backup files by adding a '~' to the end of the file name. You may have problems if you use Emacs's facility for numbered backup files, however, since then there is a '~' in the middle of the file name, which will need to be quoted when used in the shell.

Closures or repeated matches

The extended globbing symbols '`#`' and '`##`', when they occur in a pattern, are equivalent to '`*`' and '`+`' in extended regular expressions: '`#`' allows the previous pattern to match any number of times, including zero, while with '`##`' it must match at least once. Note that this pattern does not extend beyond two hashes — there is no special symbol '`###`', which is not recognised as a pattern at all.

The 'previous pattern' is the smallest possible item which could be considered a complete pattern. Very often it is something in parentheses, but it could be a group in square or angle brackets, or a single ordinary character. Note particularly that in

```
# fails
[[ foofoo = foo# ]]
```

the test fails, because the '`#`' only refers to the final '`o`', not the entire string. What you need is

```
# succeeds
[[ foofoo = (foo)# ]]
```

It might worry you that '`#`' also introduces comments. Since a well-formatted pattern never has '`#`' at the start, however, this isn't a problem unless you expect comments to start in the middle of a word. It turns out that doesn't even happen in other shells — '`#`' must be at the start of a line, or be unquoted and have space in front of it, to be recognised as introducing a comment. So in fact there is no clash at all here. There is, of course, a clash

if you expect '`.#foo.c.1.131`' (probably a file produced by the version control system CVS while attempting to resolve a conflict) to be a plain string, hence the dependence on the `EXTENDED_GLOB` option.

That's probably all you need to know; the '`#`' operators are generally much easier to understand than the exclusion operators. Just in case you are confused, I might as well point out that repeating a *pattern* is not the same as repeating a *string*, so

```
[[ onetwothreetwoone = (one|two|three)## ]]
```

successfully matches; the string is different for each repetition of the pattern, but that doesn't matter.

We now have enough information to construct a list of correspondences between zsh's normal pattern operators and the ksh ones, available with `KSH_GLOB`. Be careful with '`!(...)`'; it seems to have a slightly different behaviour to the zsh near-equivalent. The following table is lifted directly from the zsh FAQ.

```
-------------------------------------------------------------------
      ksh                zsh          Meaning
     ------             ------        ---------
     !(foo)              ^foo         Anything but foo.
                 or   foo1~foo2       Anything matching foo1 but foo2.
@(foo1|foo2|...)  (foo1|foo2|...)     One of foo1 or foo2 or ...
     ?(foo)            (foo|)         Zero or one occurrences of foo.
     *(foo)            (foo)#         Zero or more occurrences of foo.
     +(foo)            (foo)##        One or more occurrences of foo.
-------------------------------------------------------------------
```

In both languages, the vertical bar for alternatives can appear inside any set of parentheses. Beware of the precedences of `^foo` and '`foo1~foo2`'; surround them with parentheses, too, if necessary.

5.9.5 Recursive globbing

One of the most used special features of zsh, and one I've already used a couple of times in this section, is recursive globbing, the ability to match any directory in an arbitrarily deep (or, as we say in English, tall) tree of directories. There are two forms: '`**/`' matches a set of directories to any depth, including the top directory, what you get by replacing '`**/`' by '`./`, i.e. `**/foo` can match `foo` in the current directory, but also `bar/foo`, `bar/bar/bar/foo`, `bar/bar/bar/poor/little/lambs/foo` nad so on. '`***/`' does the same, but follows symbolic links; this can land you in infinite loops if the link points higher up in the same directory hierarchy — an odd thing to do, but it can happen.

The '`**/`' or '`***/`' can't appear in parentheses; there's no way of specifying them as alternatives. As already noticed, however, the precedence of the exclusion operator '`~`' provides a useful way of removing matches you don't want. Remember, too, the recursion operators don't need to be at the start of the pattern:

```
print ~/**/*.txt
```

prints the name of all the files under your home directory ending with '`.txt`'. Don't expect it to be particularly fast; it's not as well optimised as the standard UNIX command `find`,

although it is a whole lot more convenient. The traditional way of searching a file which may be anywhere in a directory tree is along the lines of:

```
find ~/src -name '*.c' -print | xargs grep pattern
```

which is horrendously cumbersome. What's happening is that `find` outputs a newline-separated list of all the files it finds, and `xargs` assembles these as additional arguments to the command '`grep pattern`'. It simplifies in zsh to the much more natural

```
grep pattern ~/src/**/*.c
```

In fact, strictly speaking you probably ought to use

```
find ~/src -name '*.c' -print0 | xargs -0 grep pattern
```

for the other form — this passes null-terminated strings around, which is safer since any character other than a NUL or a slash can occur in a filename. But of course you don't need that now.

Do remember that this includes the current directory in the search, so in that last example '`foo.c`' in the directory where you typed the command would be searched. This isn't completely obvious because of the '`/`' in the pattern, which erroneously seems to suggest at least one directory.

It's a little known fact that this is a special case of a more general syntax, '(*pat*`/`)`#`'. This syntax isn't perfect, either; it's the only time where a '`/`' can usefully occur in parentheses. The pattern *pat* is matched against each directory; if it succeeds, *pat* is matched against each of the subdirectories, and so on, again to arbitrary depth. As this uses the character '`#`', it requires the `EXTENDED_GLOB` option, which the more common syntax doesn't, since no-one would write two `*`'s in a row for any other reason.

You should consider the '`/`)' to be in effect a single pattern token; for example in

```
% print (F*|B*/)#*.txt
FOO/BAR/thingy.txt
```

both '`F*`' and '`B*`' are possible directory names, not just the '`B*`' next to the slash. The difference between '`#`' and '`##`' is respected here — with the former, zero occurrences of the pattern may be matched (i.e. '`*.txt`'), while with the latter, at least one level of subdirectories is required. Thus '`(*/)##*.txt`' is equivalent to '`*/**/*.txt`', except that the first '`*`' in the second pattern will match a symbolic link to a directory; there's no way of forcing the other syntax to follow symbolic links.

Fairly obviously, this syntax is only useful with files. Other uses of patterns treat slashes as ordinary characters and '`**`' or '`***`' the same as a single '`*`'. It's not an error to use multiple '`*`'s, though, just pointless.

5.9.6 Glob qualifiers

Another very widely used zsh enhancement is the ability to select types of file by using 'glob qualifiers', a group of (rather terse) flags in parentheses at the end of the pattern. Like

recursive globbing, this feature only applies for filename generation in the command line (including an array assignment), not for other uses of patterns.

This feature requires the BARE_GLOB_QUAL option to be turned on, which it usually is; the name implies that one day there may be another, perhaps more ksh-like, way of doing the same thing with a more indicative syntax than just a pair of parentheses.

File types

The simplest glob qualifiers are similar to what the completion system appends at the end of file names when the LIST_TYPES option is on; these are in turn similar to the indications used by '`ls -F`'. So

```
% print *(.)
file1 file2 cmd1 cmd2
% print *(/)
dir1 dir2
% print *(*)
cmd1 cmd2
% print *(@)
symlink1 symlink2
```

where I've invented unlikely filenames with obvious types. `file1` and `file2` were supposed to be just any old file; `(.)` picks up those but also executable files. Sockets `(=)`, named pipes `(p)`, and device files `(%)` including block `(%b)` and character `(%c)` special files are the other types of file you can detect.

Associated with type, you can also specify the number of hard links to a file: `(l2)` specifies exactly 2 links, `(l+3)` more than 3 links, `(l-5)` fewer than 5.

File permissions

Actually, the `(*)` qualifier really applies to the file's permissions, not it's type, although it does require the file to be an executable non-special file, not a directory nor anything wackier. More basic qualifiers which apply just to the permissions of the files are `(r)`, `(w)` and `(x)` for files readable, writeable and executable by the owner; `(R)`, `(W)` and `(X)` correspond to those for world permissions, while `(A)`, `(I)` and `(E)` do the job for group permissions — sorry, the Latin alphabet doesn't have middle case. You can speciy permissions more exactly with '`(f)`' for file permissions: the expression after this can take various forms, but the easiest is probably a delimited string, where the delimiters work just like the arguments for parameter flags and the arguments, separated by commas, work just like symbolic arguments to `chmod`; the example from the manual,

```
print *(f:gu+w,o-rx:)
```

picks out files (of any type) which are writeable by the owner ('user') and group, and neither readable nor executable by anyone else ('other').

File ownership

You can match on the other three mode bits, setuid ((s)), setgid ((S)) and sticky ((t)), but I'm not going to go into what those are if you don't know; your system's manual page for `chmod` may (or may not) explain.

Next, you can pick out files by owner; `(U)` and `(G)` say that you or your group, respectively, owns the file — really the effective user or group ID, which is usually who you are logged in as, but this may be altered by tricks such as a programme running setuid or setgid (the things I'm not going to explain). More generally, `u0` says that the file is owned by root and `(u501)` says it is owned by user ID 501; you can use names if you delimiit them, so `(u:pws:)` says that the owner must be user `pws`; similarly for groups with `(g)`.

File times

You can also pick files by modification ((m)) or access ((a)) time, either before ((-)), at, or after ((+)) a specific time, which may be measured in days (the default), months ((M)), weeks ((w)), hours ((h)), minutes ((m)) or seconds ((s)). These must appear in the order `m` or `a`, optional unit, optional plus or minus, number. Hence:

```
print *(m1)
```

Files that were modified one day ago — i.e. less than 48 but more than 24 hours ago.

```
print *(aw-1)
```

Files accessed within the last week, i.e. less than 7 days ago.

In addition to `(m)` and ((a)), there is also `(c)`, which is sometimes said to refer to file creation, but it is actually something a bit less useful, namely *inode* change. The inode is the structure on disk where UNIX-like filing systems record the information about the location and nature of the file. Information here can change when some aspect of the file information, such as permissions, changes.

File size

The qualifier `(L)` refers to the file size ('L' is actually for length), by default in bytes, but it can be in kilobytes `(k)`, megabytes `(m)`, or 512-byte blocks `(p, unfortunately)`. Plus and minus can be used in the same way as for times, so

```
print *(Lk3)
```

gives files 3k large, i.e. larger than 2k but smaller than 4k, while

```
print *(Lm+1)
```

gives files larger than a megabyte.

Note that file size applies to directories, too, although it's not very useful. The size of directories is related to the number of slots for files currently available inside the directory

(at the highest level, i.e. not counting children of children and deeper). This changes automatically if necessary to make more space available.

File matching properties

There are a few qualifiers which affect option settings just for the match in question: `(N)` turns on `NULL_GLOB`, so that the pattern simply disappears from the command line if it fails to match; `(D)` turns on `GLOB_DOTS`, to match even files beginning with a '.', as described above; `(M)` or `(T)` turn on `MARK_DIRS` or `LIST_TYPES`, so that the result has an extra character showing the type of a directory only (in the first case) or of any special file (in the second); and `(n)` turns on `NUMERIC_GLOB_SORT`, so that numbers in the filename are sorted arithmetically — so `10` comes after `1A`, because the 1 and 10 are compared before the next character is looked at.

Other than being local to the pattern qualified, there is no difference in effect from setting the option itself.

Combining qualifiers

One of the reasons that some qualifiers have slightly obscure syntax is that you can chain any number of them together, which requires that the file has all of the given properties. In other words '`*(UWLk-10)`' are files owned by you, world writeable and less than 10k in size.

You can negate a set of qualifiers by putting '^' in front of those, so '`*(ULk-10^W)`' would specify the corresponding files which were not world writeable. The '^' applies until the end of the flags, but you can put in another one to toggle back to assertion instead of negation.

Also, you can specify alternatives; '`*(ULk-10,W)`' are files which either are owned by you and are less than 10k, or are world writeable — note that the 'and' has higher precedence than the 'or'.

You can also toggle whether the assertions or negations made by qualifiers apply to symbolic links, or the files found by following symbolic links. The default is the former — otherwise the `(@)` qualifier wouldn't work on its own. By preceding qualifiers with `-`, they will follow symbolic links. So `*(-/)` matches all directories, including those reached by a symbolic link (or more than one symbolic link, up to the limit allowed by your system). As with '^', you can toggle this off again with another one '-'. To repeat what I said in chapter 3, you can't distinguish between the other sort of links, hard links, and a real file entry, because a hard link just supplies an alternative but equivalent name for a file.

There's a nice trick to find broken symlinks: the pattern '`**/*(-@)`'. This is supposed to follow symlinks; but that '`@`' tells it to match only on symlinks! There is only one case where this can succeed, namely where the symlink is broken. (This was pointed out to me by Oliver Kiddle.)

Sorting and indexing qualifiers

Normally the result of filename generation is sorted by alphabetic order of filename. The globbing flags `(o)` and `(O)` allow you to sort in normal or reverse order of other things: `n` is for names, so `(on)` gives the default behaviour while `(On)` is reverse order; `L`, `l`, `m`, `a` and `c` refer to the same thing as the normal flags with those letters, i.e. file size, number of links,

and modification, access and inode change times. Finally, `d` refers to subdirectory depth; this is useful with recursive globbing to show a file tree ordered depth-first (subdirectory contents appear before files in any given directory) or depth-last.

Note that time ordering produces the most recent first as the standard ordering (`(om)`, etc.), and oldest first as the reverse ordering `(OM)`, etc.). With size, smallest first is the normal ordering.

You can combine ordering criteria, with the most important coming first; each criterion must be preceded by `o` or `O` to distinguish it from an ordinary globbing flag. Obviously, `n` serves as a complete discriminator, since no two different files can have the same name, so this must appear on its own or last. But it's a good idea, when doing depth-first ordering, to use `odon`, so that files at a particular depth appear in alphabetical order of names. Try

```
print **/*(odon)
```

to see the effect, preferably somewhere above a fairly shallow directory tree or it will take a long time.

There's an extra trick you can play with ordered files, which is to extract a subset of them by indexing. This works just like arrays, with individual elements and slices.

```
print *([1])
```

This selects a single file, the first in alphabetic order since we haven't changed the default ordering.

```
print *(om[1,5])
```

This selects the five most recently modified files (or all files, if there are five or fewer). Negative indices are understood, too:

```
print *(om[1,-2])
```

selects all files but the oldest, assuming there are at least two.

Finally, a reminder that you can stick modifiers after qualifiers, or indeed in parentheses without any qualifiers:

```
print **/*(On:t)
```

sorts files in subdirectories into reverse order of name, but then strips off the directory part of that name. Modifiers are applied right at the end, after all file selection tasks.

Evaluating code as a test

The most complicated effect is produced by the `(e)` qualifer. which is followed by a string delimited in the now-familiar way by either matching brackets of any of the four sorts or a pair of any other characters. The string is evaluated as shell code; another layer of quotes is stripped off, to make it easier to quote the code from immediate expansion. The expression

is evaulated separately for each match found by the other parts of the pattern, with the parameter `$REPLY` set to the filename found.

There are two ways to use `(e)`. First, you can simply rely on the return code. So:

```
print *(e:'[[ -d $REPLY ]]':)
print *(/)
```

are equivalent. Note that quotes around the expression, which are necessary in addition to the delimiters (here '`:`') for expressions with special characters or whitespace. In particular, `$REPLY` would have been evaluated too early — before file generation took place — if it hadn't been quoted.

Secondly, the function can alter the value of `$REPLY` to alter the name of the file. What's more, the expression can set `$reply` (which overrides the use of `$REPLY`) to an array of files to be inserted into the command line; it may be any size from zero items upward.

Here's the example in the manual:

```
print *(e:'reply=(${REPLY}{1,2})':)
```

Note the string is delimited by colons *and* quoted. This takes each file in the current directory, and for each returns a match which has two entires, the filename with '`1`' appended and the filename with '`2`' appended.

For anything more complicated than this, you should write a shell function to use `$REPLY` and set that or `$reply`. Then you can replace the whole expression in quotes with that name.

5.9.7 Globbing flags: alter the behaviour of matches

Another `EXTENDED_GLOB` features is 'globbing flags'. These are a bit like the flags that can appear in perl regular expressions; instead of making an assertion about the type of the resulting match, like glob qualifiers do, they affect the way the match is performed. Thus they are available for all uses of pattern matching — though some flags are not particularly useful with filename generation.

The syntax is borrowed from perl, although it's not the same: it looks like '`(#X)`', where `X` is a letter, possibily followed by an argument (currently only a number and only if the letter is '`a`'). Perl actually uses '`?`' instead of '`#`'; what these have in common is that they can't appear as a valid pattern characters just after an open parenthesis, since they apply to the pattern before. Zsh doesn't have the rather technical flags that perl does (lookahead assertions and so on); not surprisingly, its features are based around the shortcuts often required by shell users.

Mixed-case matches

The simplest sort of globbing flag will serve as an example. You can make a pattern, or a portion of a pattern, match case-insensitively with the flag `(#i)`:

```
[[ FOO = foo ]]
[[ FOO = (#i)foo ]]
```

Assuming you have EXTENDED_GLOB set so that the '#' is an active pattern character, the first match fails while the second succeeds. I mentioned portions of a pattern. You can put the flags at any point in the pattern, and they last to the end either of the pattern or any enclosing set of parentheses, so in

```
[[ FOO = f(#i)oo ]]
[[ FOO = F(#i)oo ]]
```

once more the first match fails and the second succeeds. Alternatively, you can put them in parentheses to limit their scope:

```
[[ FOO = ((#i)fo)o ]]
[[ FOO = ((#i)fo)O ]]
```

gives a failure then a success again. Note that you need extra parentheses; the ones around the flag just delimit that, and have no grouping effect. This is different from Perl.

There are two flags which work in exactly the same way: (#l) says that only lowercase letters in the pattern match case-insensitively; uppercase letters in the pattern only match uppercase letters in the test string. This is a little like Emacs' behaviour when searching case insensitvely with the case-fold-search option variable set; if you type an uppercase character, it will look only for an uppercase character. However, Emacs has the additional feature that from that point on the whole string becomes case-sensitive; zsh doesn't do that, the flag applies strictly character by character.

The third flag is (#I), which turns case-insensitive matching off from that point on. You won't often need this, and you can get the same effect with grouping — unless you are applying the case-insensitive flag to multiple directories, since groups can't span more than one directory. So

```
print (#i)/a*/b*/(#I)c*
```

is equivalent to

```
print /[aA]*/[bB]*/c*
```

Note that case-insensitive searching only applies to characters not in a special pattern of some sort. In particular, ranges are not automatically made case-insensitive; instead of '(#i)[ab]*', you must use '[abAB]*'. This may be unexpected, but it's consistent with how other flags, notably approximation, work.

You should be careful with matching multiple directories case-insensitively. First,

```
print (#i)~/.Z*
```

doesn't work. This is due to the order of expansions: filename expansion of the tilde happens before pattern matching is ever attempted, and the '~' isn't at the start where filename expansion needs to find it. It's interpreted as an empty string which doesn't match '/.Z*', case-insensitively — in other words, it will match any empty string.

Hence you should put '(#i)' and any other globbing flags after the first slash — unless, for some reason, you *really* want the expression to match '/Home/PWS/' etc. as well as '/home/pws'.

Second,

```
print (#i)$HOME/.Z*
```

does work — prints all files beginning '.Z' or '.z' in your home directory — but is inefficient. Assume $HOME expands to my home directory, /home/pws. Then you are telling the shell it can match in the directories '/Home/PWS/', '/HOME/pWs' and so on. There's no quick way of doing this — the shell has to look at every single entry first in '/' and then in '/home' (assuming that's the only match at that level) to check for matches. In summary, it's a good idea to use the fact that the flag doesn't have to be at the beginning, and write this as:

```
print ~/(#i).Z*
```

Of course,

```
print ~/.[zZ]*
```

would be easier and more standard in this oversimplified example.

On Cygwin, a UNIX-like layer running on top of, uh, a well known graphical user interface claiming to be an operating system, filenames are usually case insensitive anyway. Unfortunately, while Cygwin itself is wise to this fact, zsh isn't, so it will do all that extra searching when you give it the (#i) flag with an otherwise explicit string.

A piece of good news, however, is that matching of uppercase and lowercase characters will handle non-ASCII character sets, provided your system handles locales, (or to use the standard hieroglyphics, 'i18n' — count the letters between 'i' and 'n' in 'internationalization', which may not even be a word anyway, and wince). In that case you or your system administrator or the shell environment supplied by your operating system vendor needs to set $LC_ALL or $LC_CTYPE to the appropriate locale – C for the default, en for English, uk for Ukrainian (which I remember because it's confusing in the United Kingdom), and so on.

'Backreferences'

The feature labelled as 'backreferences' in the manual isn't really that at all, which is my fault. Many regular expression matchers allow you to refer back to bits already matched. For example, in Perl the regular expression '([A-Z]{3})$1' says 'match three uppercase characters followed by the same three characters again. The '$1' is a backreference.

Zsh has a similar feature, but in fact you can't use it while matching a single pattern; it just makes the characters matched by parentheses available after a successful complete match. In this, it's a bit more like Emacs's match-beginning and match-end functions.

You have to turn it on for each pattern with the globbing flag '(#b)'. The reason for this is that it makes matches involving parentheses a bit slower, and most of the time you use parentheses just for ordinary filename generation where this feature isn't useful. Like most of the other globbing flags, it can have a local effect: only parentheses after the flag produce backreferences, and the effect is local to enclosing parentheses (which don't feel the effect themselves). You can also turn it off with '(#B)'.

What happens when a pattern with active parentheses matches is that the elements of the array $match, $mbegin and $mend are set to reflect each active parenthesis in turn —

names inspired by the corresponding Emacs feature. The string matching the first pair of parentheses is stored in the first element of `$match`, its start position in the string is stored in the first element of `$mbegin`, and its end position in the string `$mend`. The same happens for later matched parentheses. The parentheses around any globbing flags do not count.

`$mbegin` and `$mend` use the indexing convention currently in effect, i.e. zero offset if `KSH_ARRAYS` is set, unit offset otherwise. This means that if the string matched against is stored in the parameter `$teststring`, then it will always be true that `${match[1]}` is the same string as `${teststring[${mbegin[1]},${mend[1]}]}`. and so on. (I'm assuming, as usual, that `KSH_ARRAYS` isn't set.) Unfortunately, this is different from the way the `E` parameter flag works — that substitutes the character after the end of the matched substring. Sorry! It's my fault for not following that older convention; I thought the string subscripting convention was more relevant.

An obvious use for this is to match directory and non-directory parts of a filename:

```
local match mbegin mend
if [[ /a/file/name = (#b)(*)/([^/]##) ]]; then
  print -l ${match[1]} ${match[2]}
fi
```

prints '`/a/file`' and '`name`'. The second parenthesis matches a slash followed by any number of characters, but at least one, which are not slashes, while the first matches anything — remember slashes aren't special in a pattern match of this form. Note that if this appears in a function, it is a good idea to make the three parameters local. You don't have to clear them, or even make them arrays. If the match fails, they won't be touched.

There's a slightly simpler way of getting information about the match: the flag `(#m)` puts the matched string, the start index, and the index for the *whole* match into the scalars `$MATCH`, `$MBEGIN` and `$MEND`. It may not be all that obvious why this is useful. Surely the whole pattern always matches the whole string? Actually, you've already seen cases where this isn't true for parameter substitutions:

```
local MATCH MBEGIN MEND string
string=aLOha
: ${(S)string##(#m)([A-Z]##)}
```

You'll find this sets `$MATCH` to `LO`, `$MBEGIN` to 2 and `$MEND` to 3. In the parameter expansion, the `(S)` is for matching substrings, so that the '`##`' match isn't anchored to the start of `$string`. The pattern is `(#m)([A-Z]##)`, which means: turn on full-match backreferencing and match any number of capital letters, but at least one. This matches `LO`. Then the match parameters let you see where in the test parameter the match occurred.

There's nothing to stop you using both these types of backreferences at once, and you can specify multiple globbing flags in the short form '`(#bm)`'. This will work with any combination of flags, except that some such as '`(#bB)`' are obviously silly.

Because ordinary globbing produces a list of files, rather than just one, this feature isn't very useful and is turned off. However, it *is* possible to use backreferences in global substitutions and substitutions on arrays; here are both at once:

```
% array=(mananan then in gone June)
% print ${array//(#m)?n/${(C)MATCH[1]}n}
mAnAnAn thEn In gOne JUne
```

The substitution occurs separately on each element of the array, and at each match in each element `$MATCH` gets set to what was matched. We use this to capitalize every character that is followed by a lowercase 'n'. This will work with the `(#b)` form, too. The perl equivalent of this is:

```
% perl -pe 's/.n/\u$&/g' <<<$array
mAnAnAn thEn In gOne JUne
```

(People sometimes say Perl has a difficult syntax to understand; I hope I'm convincing you how naive that view is when you have zsh.)

Now I can convince you of one point I made about excluded matches above:

```
% [[ foo = (#b)(^foo)* ]]  && print $match
fo
```

As claimed, the process of making the longest possible match, then backtracking from the end until the whole thing is successful, leads to the '`(^foo)`' matching '`fo`'.

Approximate matching

To my knowledge, zsh is the first command line interpreter to make use of approximate matching. This is very useful because it provides the shell with an easy way of correcting what you've typed. First, some basics about what I mean by 'approximate matching'.

There are four ways you can make a mistake in typing. You can leave out a letter which should be there; you can insert a letter which shouldn't; you can type one letter instead of another; and you can transpose two letters. The last one involves two different characters, so some systems for making approximate matches count it as two different errors; but it's a particularly common one when typing, and quite useful to be able to handle as a single error. I know people who even have '`mkae`' aliased to '`make`'.

You can tell zsh how many errors you are willing to allow in a pattern match by using, for example `(#a1)`, which says only a single error allowed. The rules for the flag are almost identical to those for case-insensitive matching, in particular for scoping and the way approximate matching is handled for a filename expansion with more than one directory. The number of errors is global; if the shell manages to match a directory in a path with an error, one fewer error is allowed for the rest of the path. You can specify as many errors as you like; the practical limit is that with too many allowed errors the pattern will match far too many strings. The shell doesn't have a particularly nifty way of handling approximate matching (unlike, for example, the program `agrep`), but you are unlikely to encounter problems if the number of matches stays in a useful range.

The fact that the error count applies to the whole of a filename path is a bit of a headache, actually, because we have to make sure the shell matches each directory with the minimum number of errors. With a single pattern, the shell doesn't care as long as it doesn't use up all the errors it has, while with multiple directories if it uses up the errors early on, it may fail to match something it should match. But you don't have to worry about that; this explanation is just to elicit sympathy.

So the pattern `(#a1)README` will match `README`, `READ.ME`, `READ_ME`, `LEADME`, `REDME`, `READEM`, and so on. It will not match `_README_`, `ReadMe`, `READ` or `AAREADME`. However, you can combine it with case-insensitivity, for which the short form `(#ia1)README` is allowed,

and then it will match `ReadMe`, `Read.Me`, `read_me`, and so on. You can consider filenames with multiple directories as single strings for this purpose — with one exception, that '`foo/bar`' and '`fo/obar`' are two errors apart, not one. Because the errors are counted separately in each directory, you can't transpose the '`/`' with another character. This restriction doesn't apply in other forms of pattern matching where `/` is not a special character.

Another common feature with case-insensitive matching is that only the literal parts of the string are handled. So if you have '`[0-9]`' in a pattern, that character must match a decimal digit even if approximation is active. This is often useful to impose a particular form at key points. The main difficulty, as with the '`/`' in a directory, is that transposing with another character is not allowed, either. In other words, '`(#a1)ab[0-9]`' will fail to match '`a1b`'; it will match with two errors, by removing the '`b`' before the digit and inserting it after.

As an example of what you can do with this feature, here is a simple function to correct misspelled filenames.

```
emulate -LR zsh
setopt extendedglob

local file trylist
integer approx max_approx=6

file=$1

if [[ -e $file ]]; then
  # no correction necessary
  print $file
  return
fi

for (( approx = 1; approx <= max_approx; approx++ )); do
  trylist=( (#a$approx)"$file"(N) )
  (( $#trylist )) && break
done
(( $#trylist )) || return 1

print $trylist
```

The function tries to match a file with the minimum possible number of errors, but in any case no more than 6. As soon as it finds a match, it will print it out and exit. It's still possible there is more than one match with that many errors, however, and in this case the complete list is printed. The function doesn't handle '`~`' in the filename.

It does illustrate the fact that you can specify the number of approximations as a parameter. This is purely a consequence of the fact that filename generation happens right at the end of the expansion sequence, after the parameters have already been substituted away. The numbers and the letter in the globbing flag aren't special characters, unlike the parentheses and the '`#`'; if you wanted those to be special when substituted from a parameter, you would need to set the `KSH_GLOB` flag, possibly by using the '`~`' parameter flag.

A more complicated version of that function is included with the shell distribution in the file `Completion/Base/Widget/_correct_filename`. This is designed to be used either on its own, or as part of the completion system.

Indeed, the completion system described in the next chapter is where you are most likely

to come across approximate matching, buried inside approximate completion and correction — in the first case, you tell the shell to complete what you have typed, trying to correct mistakes, and in the second case, you tell the shell that you have finished typing but possibly made some mistakes which it should correct. If you already have the new completion system loaded, you can use `^Xc` to correct a word on the command line; this is context-sensitive, so more sophisticated than the function I showed.

Anchors

The last two globbing flags are probably the least used. They are there really for completeness. They are `(#s)`, to match only at the start of a string, and `(#e)`, to match only at the end. Unlike the other flags they are purely local, just making a statement about the point where they occur in the pattern.

They correspond to the much more commonly used '^' and '$' in regular expressions. The difference is that shell patterns nearly always match a complete string, so telling the pattern that a certain point is the start or end isn't usually very useful. There are two occasions when it is. The first is when the start or end is to be matched as an alternative to something else. For example,

```
[[ $file = *((#s)|/)dirpart((#e)|/)* ]]
```

succeeds if `dirpart` is a complete path segment of `$file` — with a slash or nothing at all before and after it. Remember, once again, that slashes aren't special in pattern matches unless they're performing filename generation. The effect of these two flags isn't altered at all by their being inside another set of parentheses.

The second time these are useful is in parameter matches where the pattern is not guaranteed to match a complete string. If you use `(#s)` or `(#e)`, it will force that point to be the start or end despite the operator in use. So `${`*param*`##`*pattern*`(#e)}` will remove *pattern* from `$`*param* only if it matches the entire string: the `##` must match at the head, while the `(#e)` must match at the end.

You can get the effect with `${`*param*`:#`*pattern*`}`, and further more this is rather faster. The `:#` operator has some global knowledge about how to match; it knows that since *pattern* will match as far as it can along the test string, it only needs to try the match once. However, since '`##`' just needs to match at the head of the string, it will backtrack along the pattern, trying to match *pattern* `(#e)`, entirely heedless of the fact that the pattern itself specifically won't match if it doesn't extend to the end. So it's more efficient to use the special parameter operators whenever they're available.

5.9.8 The function `zmv`

The shell is supplied with a function `zmv`, which may have been installed into the default `$fpath`, or can be found in the source tree in the directory `Functions/Misc`. This provides a way of renaming, copying and linking files based on patterns. The idea is that you give two arguments, a pattern to match, and a string which uses that pattern. The pattern to match has backreferences turned on; these are stored in the positional parameters to make them easy to refer to. The function tries to be safe: any file whose name is not changed is simply ignored, and usually overwriting an existing file is an error, too. However, it doesn't make sure that there is a one to one mapping from source to target files; it doesn't know if the target file is supposed to be a directory (though it could be smarter about that).

In the examples, I will use the option `-n`, which forces `zmv` to print out what it will do without actually doing it. This is a good thing to try first if you are unsure.

Here's a simple example.

```
% ls
foo
% zmv -n '(*)' '${(U)1}'
mv -- foo FOO
```

The pattern matches anything in the current directory, excluding files beginning with a '.' (the function starts with an '`emulate`', so `GLOB_DOTS` is forced to be off). The complete string is stored as the first backreference, which is in turn put into `$1`. Then the second argument is used and `$1` in uppercase is substituted.

Essentials of the function

The basic code in `zmv` is very simple. It boils down to more or less the following.

```
setopt nobareglobqual extendedglob
local files pattern result f1 f2 match mbegin mend

pattern=$1
result=$2

for f1 in ${~pattern}; do
  [[ $f1 = (#b)${~pattern} ]] || continue
  set -- $match
  f2=${(e)result}
  mv -- $f1 $f2
done
```

Here's what's going on. We store the arguments as `$pattern` and `$result`. We then expand the pattern to a list of files — remember that `${~pattern}` makes the characters in `$pattern` active for the purposes of globbing. For each file we find, we match against the pattern again, but this time with backreferences turned on, so that parentheses are expanded into the array `$match`. If, for some reason, the pattern match failed this time, we just skip the file. Then we store `$match` in the positional parameters; the '`--`' for `set` and for `mv` is in case `$match[1]` begins with a '`-`'.

Then we evaluate the result, assuming that it will refer to the positional parameters. In our example, `$result` contains argument '`${(U)1}`' and if we matched '`foo`', then `$1` contains foo. The effect of '`${(e)result}`' is to perform an extra substitution on the `${(U)1}`, so `$f2` will be set to `FOO`. Finally, we use the `mv` command to do the actual renaming. The effect of the `-n` option isn't shown, but it's essentially to put a '`print`' in front of the `mv` command line.

Notice I set `nobareglobqual`, turning off the use of glob qualifiers. That's necessary because of all those parentheses; otherwise, '`(*)`' would have been interpreted as a qualifier. There is an option, `-Q`, which will turn qualifiers back on, if you need them. That's still not quite ideal, since the second pattern match, the one where we actually use the backreferences, isn't filename generation, just a test against a string, so doesn't handle glob qualifers. So in that

case the code needs to strip qualifiers off. It does this by a fairly simple pattern match which will work in simple cases, though you can confuse it if you try hard enough, particularly if you have extra parentheses in the glob qualifier.

Note also the use of '`${(e)result}`' to force substitution of parameters when `$result` is evaluated. This way of doing it safely ignores other metacharacters which may be around: all `$`-expansions, plus backquote expansion, are performed, but otherwise `$result` is left alone.

More complicated examples

`zmv` has some special handling for recursive globbing, but only with the patterns `**/` and `***/`. If you put either of these in parentheses in the pattern, they will be spotted and used in the normal way. Hence,

```
% ls test
lonely
% zmv -n '(**/)lonely' '$1solitary'
mv -- test/lonely test/solitary
```

Note that, as with other uses of '`**/`', the slash is part of the recursive match, so you don't need another one. You don't need to separate `$1` from `solitary` either, since positional parameters are a special case, but you could use '`${1}solitary`' for the look of it. Like glob qualifiers, recursive matches are handled by some magic in the function; in ordinary globbing you can't put these two forms inside parentheses.

For the lazy, the option `-w` (which means 'with wildcards') will tell `zmv` to decide for itself where all the patterns are and automatically add parentheses. The two examples so far become

```
zmv -nw '*' '${(U)1}'
zmv -nw '***/lonely' '$1solitary'
```

with exactly the same effects.

Another way of getting useful effects is to use the '`${1//foo/bar}`' substitution in the second argument. This gives you a general way of substitution bits in filenames. Often, you can then get away with having '`(*)`' as the first argument:

```
zmv '(*)' '${1//(#m)[aeiou]/${(U)MATCH}}'
```

capitalises all the vowels in all filenames in the current directory. You may be familiar with a perl script called `rename` which does tricks like this (though there's another, less powerful, programme of the same name which simply replaces strings).

The effect of `zmv`

In addition to renaming, `zmv` can be made to copy or link files. If you call it `zcp` or `zln` instead of `zmv`, it will have those effects, and in the case of `zln` you can use the option `-s` to create symbolic links, just as with `ln`. Beware the slightly confusing behaviour of symbolic links containing relative directories, however.

Alternatively, you can force the behavour of `zmv`, `zcp` and `zln` just by giving the options `-M`, `-C` or `-L` to the function, whatever it is called. Or you can use '`-p` *prog*' to execute `prog` instead of `mv`, `cp` or `ln`; *prog* should be able to be run as '*prog* `--` *oldname newname*', whatever it does.

The option `-i` works a bit like the same option to the basic programmes which `zmv` usually calls, prompting you before any action — in this case, not just overwriting, but any action at all. Likewise, `-f` tells `zmv` to force overwriting of files, which it will usually refuse to do because of the potential dangers. Although many versions of `mv` etc. take this option, some don't, so it's not passed down; instead there's a generic way of passing down options to the programmes executed, using `-o` followed by a string. For example,

```
% ls
foo
% zmv -np frud -o'-a -b' '(*)' '${(U)1}'
frud -a -b -- foo FOO
```

Chapter 6

Completion, old and new

Completion of command arguments is something zsh is particularly good at. The simplest case is that you hit <TAB>, and the shell guesses what has to go there and fills it in for you:

```
% ls
myfile  theirfile  yourfile
% cat t<TAB>
```

expands the command line to

```
% cat theirfile
```

and you only had to type the initial letter, then TAB.

In the early days when this feature appeared in the C shell, only filenames could be completed; there were no clever tricks to help you if the name was ambiguous, it simply printed the unambiguous part and beeped so that you had to decide what to do next. You could also list possible completions; for some reason this became attached to the ^D key in csh, which in later shells with Emacs-like bindings also deletes the next character, so that history has endowed zsh, like other shells, with the slightly odd combined behaviour:

```
% cat yx
```

Now move the cursor back one character onto the x and hit ^D twice and you see: yourfile. That doesn't work if you use vi-like bindings, or, obviously, if you've rebound ^D.

Next, it became possible to complete other items such as names of users, commands or hosts. Then zsh weighed in with menu completion, so you could keep on blindly hitting <TAB> until the right answer appeared, and never had to type an extra character yourself.

The next development was tcsh's, and then zsh's, programmable completion system; you could give instructions to the shell that in certain contexts, only certain items should be completed; for example, after cd, you would only want directories. In tcsh, there was a command called complete; each 'complete ...' statement defined the completion for the arguments of a particular command, such as cd; the equivalent in zsh is compctl, which was inspired by complete but is different in virtually every significant detail. There is a perl script lete2ctl in the Misc directory of the shell distribution to help you convert from

the tcsh to the zsh formats. You put a whole series of `compctl` commands into `.zshrc`, and everything else is done by the shell.

Zsh's system has become more and more sophisticated, and in version 3.1.6 a new completion system appeared which is supposed to do everything for you: you simply call a function, `compinit`, from an initialization file, after which zsh knows, for example, that `gunzip` should be followed by files ending in `.gz`. The new system is based on shell functions, an added bonus since they are extremely flexible and you already know the syntax. However, given the complexity it's quite difficult to get started writing your own completions now, and hard enough to know what to do to change the settings the way you like. The rest of the chapter should help.

I shall concentrate on the new completion system, which seems destined to take over completely from the old one eventually, now that the 3.1 release series has become the 4.0 production release. The old **compctl** command is still available, and old completion definitions will remain working in future versions of zsh — in fact, on most operating systems which support dynamically linked libraries the old completion system is in a different file, which the shell loads when necessary, so there's very little overhead for this.

The big difference in the new system is that, instead of everything being set up once and for all when the shell starts, various bits of shell code are called after you hit `<TAB>`, to generate the completions there and then. There's enough new in the shell that all those unmemorable options to `compctl` ('`-f`' for files '`-v`' for variables and so on) can be replaced by commands that produce the list of completions directly; the key command in this case is called '`compadd`', which is passed this list and decides what to use to complete the word on the command line. So the simplest possible form of new completion looks roughly like this:

```
# tell the shell that the function mycompletion can do completion
# when called by the widget name my-completion-widget, and that
# it behaves like the existing widget complete-word
zle -C my-completion-widget .complete-word mycompletion

# define a key that calls the completion widget
bindkey '^x^i' my-completion-widget

# define the function that will be called
mycompletion() {
  # add a list of completions
  compadd alpha bravo charlie delta
}
```

That's very roughly what the completion system is doing, except that the function is called `_main_complete` and calls a lot of other functions to do its dirty work based on the context where completion was called (all the things that `compctl` used to do), and the widgets are just the old completion widgets ('`expand-or-complete`' etc.) redefined and still bound to all the original keys. But, in case you hadn't guessed, there's more to it than that.

Here's a plan for the sections of this chapter.

1. A broad description of completion and expansion, applying equally to old and new completion.
2. How to configure completion using shell options. Most of this section applies to old completion, too, although I won't explicitly flag up any differences. After this, I shall leave the `compctl` world behind.

3. How to start up new completion.
4. The basics of how the new completion system works.
5. How to configure it using the new 'zstyle' builtin.
6. Separate commands which do something other than the usual completion system, as well as some other editing widgets that have to do with completion.
7. Matching control, a powerful way of deciding such things as whether to complete case-insensitively, to allow insertion of extra parts of words before punctuation characters, or to ignore certain characters in the word on the command line.
8. How to write your own completion functions; you won't need to have too solid an understanding of all the foregoing just to do simple completions, but I will gradually introduce the full baroque splendour of how to make tags and styles work in your own functions, and how to make completion do the work of handling command arguments and options.
9. Ends the chapter gracefully, on the old 'beginning, middle, end' principle.

6.1 Completion and expansion

More things than just completion happen when you hit tab. The first thing that zsh tries to do is expand the line. Expansion was covered in a previous chapter: basically all the things described there are possible candidates for expanding in-line by the editor. In other words, history substitutions with bangs, the various expansions using '$' or backquote, and filename generation (globbing) can all take place, with the result replacing what was there on the command line:

```
% echo $PWD<TAB>
  ->    echo /home/pws/zsh/projects/zshguide
% echo `print $ZSH_VERSION`<TAB>
  ->    echo 3.1.7
% echo !!<TAB>
  ->    echo echo 3.1.7
% echo ~/.z*<TAB>
  ->    echo /home/pws/.zcompdump /home/pws/.zlogout
        /home/pws/.zshenv /home/pws/.zshrc
```

Note that the '~' also gets expanded in this case.

This is often a good time to remember the 'undo' key, '^_' or '^Xu'; typing this will restore what was there before the expansion if you don't like the result. Many keyboards have a quirk that what's described as '^_' should be typed as control with slash, which you'd write '^/' except unfortunately that does something else; this is not zsh's fault. There's another half-exception, namely filename generation: paths like '~/file' don't get expanded, because you usually know what they refer to and it's usually convenient to leave them for use in completion. However, the '=cmdname' form does get expanded, unless you have NO_EQUALS set.

In fact, deciding whether expansion or completion takes place can sometimes be tricky, since things that would be expanded if they were complete, may need to be completed first; for example $PAT should probably be completed to $PATH, but it's quite possible

there is a parameter `$PAT` too. You can decide which, if you prefer. First, the commands `expand-word`, bound to '`^X*`', and the corresponding command for listing what would be expanded, `list-expand`, bound to '`^Xg`', do expansion only — all possible forms except alias expansion, including turning '`~/file`' into a full path.

From the other point of view, you can use commands other than `expand-or-complete`, the one bound by default to `<TAB>`, to perform only completion. The basic command for this is `complete-word`, which is not bound by default. It is quite sensible to bind this to '`^I`' (i.e. `<TAB>`) if you are happy to use the separate commands for expansion, i.e.

```
# Now tab does only completion, not expansion
bindkey '^i' complete-word
```

Furthermore, if you do this and use the new completion system, then as we shall see there is a way of making the completion system perform expansion — see the description of the `_expand` completer below. In this case you have much more control over what forms of expansion are tried, and at what point, but you have to make sure you use `complete-word`, not `expand-or-complete`, else the standard expansion system will take over.

There's a close relative of `expand-or-complete`, `expand-or-complete-prefix`, not bound by default. The only difference is that it will ignore everything under and to the right of the cursor when completing. It's as if there was a space where the cursor was, with everything to be ignored shifted to the right (guess how it's implemented). Use this if you habitually type new words in the line before other words, and expect them to complete or expand on their own even before you've typed the space after them. Some other shells work this way all the time. To be more explicit:

```
% ls
filename1
% ls filex
```

Move the cursor to the `x` and hit tab. With `expand-or-complete` nothing happens; it's trying to complete a file called '`filex`' — or, with the option `COMPLETE_IN_WORD` set, it's trying to find a file whose name starts with '`file`' and ends with '`x`'. If you do

```
bindkey '^i' expand-or-complete-prefix
```

and try the same experiment, you will find the whole thing is completed to '`filename1x`', so that the '`x`' was ignored, but not removed.

One possible trap is that the listing commands, both `delete-char-or-list`, bound by default to '`^D`' in emacs mode, and `list-options`, bound by default to '`^D`' in vi insert mode and the basic command for listing completions as it doesn't have the delete-character behaviour, do not show possible expansions, so with the default bindings you can use '`^D`' to list, then hit `<TAB>` and find that the line has been completely rewritten by some expansion. Using `complete-word` instead of `expand-or-complete` will of course fix this. If you know how to write new editor widgets (chapter 4), you can make up a function which tries `list-expand`, and if that fails tries `list-options`.

There are four completion commands I haven't mentioned yet: three are `menu-complete`, `menu-expand-or-complete` and `reverse-menu-complete`, which perform menu completion, where you can cycle through all possible completions by hitting the same key. The first two correspond to `complete-word` and `expand-or-complete` respectively, while

the third has no real equivalent as it takes you backwards through a completion list. The effect of the third can't be reached just by setting options for menu completion, so it's a useful one to bind separately. I have it bound to '`\M-\C-i`', i.e. tab with the Meta key pressed down, but it's not bound by default.

The fourth is `menu-select`, which performs an enhanced form of menu completion called 'menu selection' which I'll describe below when I talk about options. You have to make sure the `zsh/complist` module is loaded to use this zle command. If you use the style, zsh should be able to load this automatically when needed, as long as you have dynamic loading, which you probably do these days.

6.2 Configuring completion using shell options

There are two main ways of altering the behaviour of completion without writing or rewriting shell functions: shell options, as introduced in chapter 2, and styles, as introduced above. I shall first discuss the shell options, although as you will see some of these refer to the styles mechanism. Setting shell options affects every single completion, unless special care has been taken (using a corresponding style for the context, or setting an option locally) to avoid that.

In addition to the options which directly affect the completion system, completion is sensitive to various other options which describe shell behaviour. For example, if the option `MAGIC_EQUAL_SUBST` is set, so that arguments of all commands looking like '`foo=~/file`' have the '~' expanded as if it was at the start of an argument, then the default completion for arguments of commands not specially handled will try to complete filenames after the '='.

Needless to say, if you write completion functions you will need to worry about a lot of other options which can affect shell syntax. The main starting point for completion chosen by context (everything except the commands for particular completions bound separately to keystrokes) is the function `_main_complete`, which includes the effect of the following lines to make sure that at least the basic options are set up within completion functions:

```
setopt glob bareglobqual nullglob rcexpandparam extendedglob unset
unsetopt markdirs globsubst shwordsplit shglob ksharrays cshnullglob
unsetopt allexport aliases errexit octalzeroes
```

but that by no means exhausts the possibilities. Actually, it doesn't include those lines: the options to set are stored in the array `$_comp_options`, with `NO_` in front if they are to be turned off. You can modify this if you find you need to (and maybe tell the maintainers, too).

By the way, if you are wondering whether you can re-use the function `_main_complete`, by binding it to a different key with slightly different completion definitions, look instead at the description of the `_generic` command widget below. It's just a front-end to `_main_complete` which allows you to have a different set of styles in effect.

6.2.1 Ambiguous completions

The largest group of options deals with what happens when a completion is ambiguous, in other words there is more than one possible completion. The seven relevant options are as follows, as copied from the FAQ; many different combinations are possible:

- with NO_BEEP set, that annoying beep goes away,
- with NO_LIST_BEEP, beeping is only turned off for ambiguous completions,
- with AUTO_LIST set, when the completion is ambiguous you get a list without having to type ^D,
- with BASH_AUTO_LIST set, the list only happens the second time you hit tab on an ambiguous completion,
- with LIST_AMBIGUOUS, this is modified so that nothing is listed if there is an unambiguous prefix or suffix to be inserted — this can be combined with BASH_AUTO_LIST, so that where both are applicable you need to hit tab three times for a listing,
- with REC_EXACT, if the string on the command line exactly matches one of the possible completions, it is accepted, even if there is another completion (i.e. that string with something else added) that also matches,
- with MENU_COMPLETE set, one completion is always inserted completely, then when you hit TAB it changes to the next, and so on until you get back to where you started,
- with AUTO_MENU, you only get the menu behaviour when you hit TAB again on the ambiguous completion.

6.2.2 ALWAYS_LAST_PROMPT

The option ALWAYS_LAST_PROMPT is set by default, and has been since an earlier 3.1 release of zsh; after listing a completion, the cursor is taken back to the line it was on before, instead of reprinting it underneath. The downside of this is that the listing will be obscured when you execute the command or produce a different listing, so you may want to unset the option. ALWAYS_LAST_PROMPT behaviour is required for menu selection to work, which is why I mention it now instead of in the ragbag below.

When you're writing your own editor functions which invoke completion, you can actually cancel the effect of this with the widget end-of-list, which you would call as zle end-of-list (it's a normal editing function, not a completion function). You can also bind it to a key to use to preserve the existing completion list. On the other hand, if you want to control the behaviour within a completion function, i.e. to decide whether completion will try to return to the prompt above the list, you can manipulate it with the last_prompt element of the $compstate associative array, so for example:

```
compstate[last_prompt]=''
```

will turn off the behaviour for the completion in progress. $compstate is the place to turn if you find yourself wanting to control completion behaviour in this much detail; see the zshcompwid manual page.

6.2.3 Menu completion and menu selection

The most significant matter decided by the options above is whether or not you are using menu completion. If you are not, you will need to type the next character explicitly when completion is ambiguous; if you are, you just need to keep hitting tab until the completion

you want appears. In the second case, of course, this works best if there are not too many possibilities. Use of AUTO_MENU or binding the menu-complete widget to a separate keystroke gives you something of both worlds.

A new variant of menu completion appeared in 3.1.6; in fact, it deserves the name menu completion rather more than the original form, but since that name was taken it is called 'menu selection'. This allows you to move the cursor around the list of completions to select one. It is implemented by a separate module, zsh/complist; you can make sure this is loaded by putting 'zmodload -i zsh/complist' in .zshrc, although it should be loaded automatically when the style menu is set as below. For it to be useful, you need two other things. The first is ALWAYS_LAST_PROMPT behaviour; this is suppressed if the whole completion list won't appear on the screen, since there's no line on the screen to go back to. However, menu selection does still work, by allowing you to scroll the list up and down. The second thing is that you need to start menu completion in any of the usual ways; menu selection is an addition to menu completion, not a replacement.

Now you should set the following style:

```
zstyle ':completion:*' menu select=<NUM>
```

If an ambiguous completion produces at least <NUM> possibilities, menu selection is started. You can understand this best by trying it. One of the completions in the list, initially the top-leftmost, is highlighted and inserted into the line. By moving the cursor in the obvious directions (with wraparound at the edges), you change both the value highlighted and the value inserted into the line. When you have the value you want, hit return, which removes the list and leaves the inserted value. Hitting ^G (the editor function send-break) aborts menu selection, removes the list and restores the command line.

Internally, zsh actually uses the parameter $MENUSELECT to supply the number and hence start menu selection. However, this is always initialised from the style as defined above, so you shouldn't set $MENUSELECT directly (unless you are using compctl, which will happily use menu selection). As with other styles, you can specify different values for different contexts; the default tag is checked if the current context does not produce a value for the style with whatever the current tag is. Note that the menu style also allows you to control whether menu completion is started at all, with or without selection; in other words, it is a style corresponding to the MENU_COMPLETE option.

There is one other additional feature when using menu selection. The zle command accept-and-infer-next-history has a different meaning here; it accepts a completion, and then tries to complete again using menu selection. This is very useful with directory hierarchies, and in combination with undo gives you a simple file browser. You need to bind it in the special keymap menuselect; for example, I use

```
bindkey -M menuselect '^o' accept-and-infer-next-history
```

because the behaviour reminds me of what is usually bound to ^O in emacs modes, namely accept-line-and-down-history. Binding it like this has no effect on ^O in the normal keymaps. Try it out by entering menu selection on a set of files including directories, and typing ^O on one of the directories. You should immediately have the contents of that directory presented for the next selection, while undo is smart enough not only to remove that selection but return to completion on the parent directory.

You can choose the manner in which the currently selected value in the completion list is highlighted using exactly the same mechanism as for specifying colours for particular types of matches; see the description of the list-colors style below.

6.2.4 Other ways of changing completion behaviour

COMPLETE_ALIASES

If you set an alias such as

```
alias pu=pushd
```

then the alias 'pu' will be expanded when the completion system is looking for the name of the command, so that it will instead find the command name 'pushd'. This is quite useful to avoid having to define extra completions for all your aliases. However, it's possible you may want to define something different for the alias than for the command it expands to. In that case, you will need to set COMPLETE_ALIASES, and to make arrangements for completing after every alias which does not already match the name of a command. Hence 'alias zcat="myzcat -dc"' will work with the option set, even if you haven't told the system about 'myzcat', while 'alias myzcat="gzip -dc"' will not work unless you do define a completion for myzcat: here 'compdef _gzip myzcat' would probably be good enough. Without the option set, it would be the other way around: the first alias would not work without the extra compdef, but the second would.

AUTO_REMOVE_SLASH

This option is turned on by default. If you complete a directory name and a slash is added — which it usually is, both to tell you that you have completed a directory and to allow you to complete files inside it without adding a '/' by hand — and the next thing you type is *not* something which would insert or complete part of a file in that directory, then the slash is removed. Hence:

```
% rmdir my<TAB>
  ->  rmdir mydir/
% rmdir mydir/<RETURN>
  ->  `rmdir mydir' executed
```

This example shows why this behaviour was added: some versions of 'rmdir' baulk at having the slash after the directory name. On the other hand, if you continued typing after the slash, or hit tab again to complete inside mydir, then the slash would remain.

This is at worst harmless under most circumstances. However, you can unset the option AUTO_REMOVE_SLASH if you don't like that behaviour. One thing that may cause slight confusion, although it is the same as with other suffixes (i.e. bits which get added automatically but aren't part of the value being completed), is that the slash is added straight away if the value is being inserted by menu completion. This might cause you to think wrongly that the completion is finished, and hence is unique when in fact it isn't.

Note that some forms of completion have this type of behaviour built in, not necessarily with a slash, when completing lists of arguments. For example, enter 'typeset ZSH_V<TAB>' and you will see 'ZSH_VERSION=' appear, in case you want to assign something to the parameter; hitting space, which is not a possible value, makes the '=' disappear. This is not controlled by the AUTO_REMOVE_SLASH option, which applies only to directories inserted by the standard filename completion system.

AUTO_PARAM_SLASH, AUTO_PARAM_KEYS

These options come into effect when completing expressions with parameter substitutions. If AUTO_PARAM_SLASH is set, then any parameter expression whose value is the name of a directory will have a slash appended when completed, just as if the value itself had been inserted by the completion system.

The behaviour for AUTO_PARAM_KEYS is a bit more complicated. Try this:

```
print ${ZSH_V<TAB>
```

You will find that you get the complete word '${ZSH_VERSION}', with the closing brace and (assuming there are no other matching parameters) a space afterwards. However, often after you have completed a parameter in this fashion you want to type something immediately after it, such as a subscript. With AUTO_PARAM_KEYS, if you type something at this point which seems likely to have to go after the parameter name, it will immediately be put there without you having to delete the intervening characters — try it with '[', for example. Note that this only happens if the parameter name and any extra bits were added by completion; if you type everything by hand, typing '[' will not have this magic effect.

COMPLETE_IN_WORD

If this is set, completion always takes place at the cursor position in the word. For example if you typed 'Mafile', went back over the 'f', and hit tab, the shell would complete 'Makefile', instead of its usual behaviour of going to the end of the word and trying to find a completion there, i.e. something matching 'Mafile*'. Some sorts of new completion (such as filename completion) seem to implement this behaviour regardless of the option setting; some other features (such as the '_prefix' completer described below) require it, so it's a good thing to set and get used to, unless you really need to complete only at the end of the word.

ALWAYS_TO_END

If this is set, the cursor is always moved to the end of the word after it is completed, even if completion took place in the middle. This also happens with menu completion.

6.2.5 Changing the way completions are displayed

LIST_TYPES

This is like the -F option to ls; files which appear in the completion listing have a trailing '/' for a directory, '*' for a regular file executable by the current process, '@' for a link, '|' for a named pipe, '%' for a character device and '#' for a block device. This option is on by default.

Note that the identifiers only appear if the completion system knows that the item is supposed to be a file. This is automatic if the usual filename completion commands are used. There is also an option -f to the builtin compadd if you write your own completion function and want to tell the shell that the values may be existing files to apply LIST_TYPES to (though no harm is caused if no such files exist).

LIST_PACKED, LIST_ROWS_FIRST

These affect the arrangement of the completion listing. With `LIST_PACKED`, completion lists are made as compact as possible by varying the widths of the columns, instead of formatting them into a completely regular grid. With `LIST_ROWS_FIRST`, the listing order is changed so that adjacent items appear along rows instead of down columns, rather like `ls`'s `-x` option.

It is possible to alter both these for particular contexts using the styles `list-packed` and `list-rows-first`. The styles in such cases always override the option; the option setting is used if no corresponding style is found.

Note also the discussion of completion groups later on: it is possible to have different types of completion appear in separate lists, which may then be formatted differently using these tag-sensitive styles.

6.3 Getting started with new completion

Before I go into any detail about new completion, here's how to set it up so that you can try it out. As I said above, the basic objects that do completions are shell functions. These are all autoloaded, so the shell needs to know where to find them via the `$fpath` array. If the shell was installed properly, and nothing in the initialization files has removed the required bits from `$fpath`, this should happen automatically. It's even possible your system sets up completion for you (Mandrake Linux 6.1 is the first system known to do this out of the box), in which case type '`which compdef`' and you should see a complete shell function — actually the one which allows you to define additional completion functions. Then you can skip the next paragraph.

If you want to load completion, try this at the command line:

```
autoload -U compinit
compinit
```

which should work silently. If not, you need to ask your system administrator what has happened to the completion functions or find them yourself, and then add all the required directories to your `$fpath`. Either they will all be in one big directory, or in a set of subdirectories with the names `AIX`, `BSD`, `Base`, `Debian`, `Redhat`, `Unix`, `X` and `Zsh`; in the second case, all the directories need to be in `$fpath`. When this works, you can add the same lines, including any modification of `$fpath` you needed, to your `.zshrc`.

You can now see if it's actually working. Type '`cd `', then `^D`, and you should be presented with a list of directories only, no regular files. If you have `$cdpath` set, you may see directories that don't appear with `ls`. As this suggests, the completion system is supplied with completions for many common (and some quite abstruse) commands. Indeed, the idea is that for most users completion just works without intervention most of the time. If you think it should when it doesn't, it may be a bug or an oversight, and you should report it.

Another example on the theme of 'it just works':

```
tar xzf archive.tar.gz ^D
```

will look inside the gzipped tar archive — assuming the GNU version of `tar`, for which the '`z`' in the first set of arguments reports that the archive has been compressed with gzip —

and give you a list of files or directories you can extract. This is done in a very similar way to normal file completion; although there are differences, you can do completion down to any directory depth within the archive. (At this point, you're supposed to be impressed.)

The completion system knows about more than just commands and their arguments, it also understands some of the shell syntax. For example, there's an associative array called `$_comps` which stores the names of commands as keys and the names of completion functions as the corresponding values. Try typing:

```
print ${_comps[
```

and then `^D`. You'll probably get a message asking if you really want to see all the possible completions, i.e. the keys for `$_comps`; if you say '`y`' you'll see a list. If you insert any of those keys, then close the braces so you have e.g. '`${_comps[mozilla]}`' and hit return, you'll see the completion function which handles that command; in this case (at the time of writing) it's `_webbrowser`. This is one way of finding out what function is handling a particular command. If there is no entry — i.e. the '`print ${_comps[mycmd]}`' gives you a blank line — then the command is not handled specially and will simply use whatever function is defined for the '`-default-`' context, usually `_default`. Usually this will just try to complete file names. You can customize `_default`, if you like.

Apart from `-default-`, some other of those keys for `_comps` also look like `-this-`: they are special contexts, places other than the arguments of a command. We were using the context called `-subscript-`; you'll find that the function in this case is called `_subscript`. Many completion functions have names which are simply an underscore followed by the command or context name, minus any hyphens. If you want a taster of how a completion function looks, try '`which _subscript`'; you may well find there are a lot of other commands in there that you don't know yet.

It's important to remember that the function found in this way is at the root of how a completion is performed. No amount of fiddling with options or styles — the stuff I'm going to be talking about for the next few sections — will change that; if you want to change the basic completion, you will just have to write your own function.

By the way, you may have old-style completions you want to mix-in — or maybe you specifically don't want to mix them in so that you can make sure everything is working with the new format. By default, the new completion system will first try to find a specific new-style completion, and if it can't it will try to find a `compctl`-defined completion for the command in question. If all that fails, it will try the usual new-style default completion, probably just filename completion. Note that specific new-style completions take precedence, which is fair enough, since if you've added them you almost certainly don't want to go back and use the old form. However, if you don't ever want to try old-style completion, you can put the following incantation in your `.zshrc`:

```
zstyle ':completion:*' use-compctl false
```

For now, that's just black magic, but later I will explain the 'style' mechanism in more detail and you will see that this fits in with the normal way of turning things off in new-style completion.

6.4 How the shell finds the right completions

6.4.1 Contexts

The examples above show that the completion system is highly context-sensitive, so it's important to know how these contexts are described. This system evolved gradually, but everything I say applies to all versions of zsh with the major version 4.

state we are at in completion, and is given as a sort of colon-separated path, starting with the least specific part. There's an easy way of finding out what context you are in: at the point where you want to complete something, instead type '^Xh', and it will tell you. In the case of the $_comps example, you will find,

```
:completion::complete:-subscript-::
```

plus a list of so-called 'tags' and completion functions, which I'll talk about later. The full form is:

```
:completion:<func>:<completer>:<command>:<argument>:<tag>
```

where the elements may be missing if they are not set, but the colons will always be there to make pattern matching easier. Here's what the bits of the context mean after the :completion: part, which is common to the whole completion system.

<func> is the name of a function from which completion is called — this is blank if it was started from the standard completion system, and only appears in a few special cases, listed in section six of this chapter.

<completer> is called 'complete' in this case: this refers to the fact that the completion system can do more than just simple completion; for example, it can do a more controlled form of expansion (as I mentioned), spelling correction, and completing words with spelling mistakes. I'll introduce the other completers later; 'complete' is the simplest one, which just does basic completion.

<command> is the name of a command or other similar context as described above, here '-subscript-'.

<argument> is most useful when <command> is the name of a real command; it describes where in the arguments to that command we are. You'll see how it works in a moment. Many of the simpler completions don't use this; only the ones with complicated option and argument combinations. You just have to find out with ^Xh if you need to know.

<tag> describes the type of a completion, essentially a way of discriminating between the different things which can be completed at the same point on the command line.

Now look at the context for a more normal command-argument completion, e.g. after cd; here you'll see the context ':completion::complete:cd::'. Here the command-name part of the context is a real command.

For something more complicated, try after 'cvs add' (it doesn't matter for this if you don't have the cvs command). You'll see a long and repetitive list of tags, for two possible contexts,

```
:completion::complete:cvs:argument-rest:
:completion::complete:cvs-add:argument-rest:
```

The reason you have both is that the 'add' is not only an argument to cvs, as the first context would suggest, it's also a subcommand in its own right, with its own arguments, and that's what the second context is for. The first context implies there might be more subcommands after 'add' and its arguments which are completely separate from them — though in fact CVS doesn't work that way, so that form won't give you any completions here.

In both, 'argument-rest' shows that completion is looking for another argument, the 'rest' indicating that it is the list of arguments at the end of the line; if position were important (see 'cvs import' for an example), the context would contain 'argument-1', or whatever. The 'cvs-add' shows how subcommands are handled, by separating with a hyphen instead of a colon, so as not to confuse the different bits of the context.

Apart from arguments to commands and subcommands, arguments to options are another frequent possibility; for an example of this, try typing ^Xh after 'dvips -o' and you will see the context ':completion::complete:dvips:option-o-1:'; this shows you are completing the first argument to dvips's -o option, (it only takes one argument) which happens to be the name of a file for output.

6.4.2 Tags

Now on to the other matter to do with contexts, tags. Let's go back and look at the output from the ^Xh help test after the cd command in full:

```
tags in context :completion::complete:cd::
  local-directories path-directories  (_alternative _cd)
```

Unlike the contexts considered so far, which tell you how completion arrived at the point it did, the tags describe the things it can complete here. In this case, there are three: directory-stack refers to entries such as '+1'; the directory stack is the set of directories defined by using the pushd command, which you can see by using the dirs command. Next, local-directories refers to subdirectories of the current working directory, while path-directories refers to any directories found by searching the $cdpath array. Each of the possible completions which the system offers belongs to one of those classes.

In parentheses, you see the names of the functions which were called to generate the completions; these are what you need to change or replace if you want to alter the basic completion behaviour. Calling functions appear on the right and called functions on the left, so that in this case the function '_cd' was the function first called to handle arguments for the cd command, fitting the usual convention. Some standard completion functions have been filtered out of this list — it wouldn't help you to know it had been through _main_complete and _complete, for example.

Maybe it's already obvious that having the system treat different types of completion in different ways is useful, but here's an example, which gives you a preview of the 'styles' mechanism, discussed later. Styles are a sort of glorified shell parameter; they are defined with the zstyle command, using a style name and possible values which may be an array; you can always define a style as an array, but some styles may simply use it as a string, joining together the arguments you gave it with spaces. You can also use the zstyle command, with different arguments, to retrieve their value, which is what the completion system itself does; there's no actual overlap with parameters and their values, so they don't get in the way of normal shell programming.

Where styles differ from parameters is that they can take different values in different contexts. The first argument to the zstyle command gives a context; when you define a

style, this argument is actually a pattern which will be matched against the current context to see if the style applies. The rule for finding out what applies is: exact string matches are preferred before patterns, and longer patterns are preferred before shorter patterns. Here's that example:

```
zstyle ':completion:*:cd:*' tag-order local-directories \
  path-directories
```

From the discussion of contexts above, the pattern will match any time an argument to the `cd` command is being completed. The style being set is called `tag-order`, and the values are the two tags valid for directories in `cd`.

The `tag-order` style determines the order in which tags are tried. The value given above means that first `local-directories` will be completed; only if none can be completed will `path-directories` be tried. You can enter the command and try this; if you don't have `$cdpath` set up you can assign '`cdpath=(~)`', which will allow '`cd foo`' to change to a directory '`~/foo`' and allow completion of directories accordingly. Go to a directory other than `~`; completion for `cd` will only show subdirectories of where you are, not those of `~`, unless you type a string which is the prefix of a directory under `~` but not your current directory. For example,

```
% cdpath=(~)
% ls -F ~
foo/    bar/
% ls -F
rod/    stick/
# Without that tag-order zstyle command, you would get...
% cd ^D
bar/    foo/    rod/    stick/
% zstyle ':completion:*:cd:*' tag-order local-directories \
   path-directories
# now you just get the local directories, if there are any...
% cd ^D
rod/    stick/
```

There's more you can do with the `tag-order` style: if you put the tags into the same word by quoting, for example "`local-directories path-directories`", then they would be tried at the same time, which in this case gives you the effect of the default. In fact, since it's too much work to know what tags are going to be available for every single possible completion, the default when there is no appropriate `tag-order` is simply to try all the tags available in the context at once; this was of course what was originally happening for completion after `cd`.

Even if there is a `tag-order` specification, any tags not specified will usually be tried all together at the end, so you could actually have missed out `path-directories` from the end of the original example and the effect would have been the same. If you don't want that to happen, you can specify a '`-`' somewhere in the list of tags, which is not used as a tag but tells completion that only the tags in the list should be tried, not any others that may be available. Also, if you don't want a particular tag to be shown you can include '`!tagname`' in the values, and all the others but this will be included. For example, you may have noticed that when completing in command position you are offered parameters to set as well as commands etc.:

```
Completing external command
```

```
tex             texhash         texi2pdf        text2sf
texconfig       texi2dvi        texindex        textmode
texdoc          texi2dvi4a2ps   texlinks        texutil
texexec         texi2html       texshow         texview
Completing parameter
TEXINPUTS                                texinputs
```

(I haven't told you how to produce those descriptions, or how to make the completions for different tags appear separately, but I will — see the descriptions of the '`format`' and '`group-name`' styles below.) If you set

```
zstyle ':completion:*:-command-:*' tag-order '!parameters'
```

then the last two lines will disappear from the completion. Of course, your completion list probably looks completely different from mine anyway. By the way, one good thing about styles is that it doesn't matter whether they're defined before or after completion is loaded, since styles are stored and retrieved by another part of the shell.

To exclude more than one tag name, you need to include the names in the same word. For example, to exclude both parameters and reserved words the value would be `'!parameters reserved-words'`, and *not* `'!parameters' '!reserved-words'`, which would try completion once with parameters excluded, then again with reserved words excluded. Furthermore, tags can actually be patterns, or more precisely any word in one of the arguments to `tag-order` may contain a pattern, which will then be tried against all the valid tags to see if it matches. It's sometimes even useful to use '`*`' to match all tags, if you are specifying a special form of one of the tags — maybe using a label, as described next — in the same word. See the manual for all the tag names understood by the supplied functions.

The `tag-order` style allows you to give tags 'labels', which are a sort of alias, instructing the completion system to use a tag under a different name. You arrange this by giving the tag followed by a colon, followed by the label. The label can also have a hyphen in front, which means that the original tag name should be put in front when the label is looked up; this is really just a way of making the names look neater. The upshot is that by using contexts with the label name in, rather than the tag name, you can arrange for special behaviour. Furthermore, you can give an alternative description for the labelled tag; these show up with the `format` style which I'll describe below (and which I personally find very useful). You put the description after another colon, with any spaces quoted. It would look like this:

```
zstyle ':completion:*:aliens:*' tag-order \
'frooble:-funny:funny\ frooble' frooble
```

which is used when you're completing for the command `aliens`, which presumably has completions tagged as '`frooble`' (if not, you're very weird). Then completion will first look up styles for that tag under the name `frooble-funny`, and if it finds completions using those styles it will list them with a description (if you are using `format`) of 'funny frooble'. Otherwise, it will look up the styles for the tag under its usual name and try completion again. It's presumably obvious that if you don't have different styles for the two labels of the tag, you get the same completions each time.

Rather than overload you with information on tags by giving examples of how to use tag labels now, I'll reserve this for the description of the `ignored-patterns` style below, which is one neat use for labels. In fact, it's the one for which it was invented; there are probably lots of other ones we haven't thought of yet.

One important note about `tag-order` which I may not have made as explicit as I should have: *it doesn't change which tags are actually valid in that completion*. Just putting a tag name into the list doesn't mean that tag name will be used; that's determined entirely by the completion functions for a particular context. The `tag-order` style simply alters the order in which the tags which *are* valid are examined. Come back and read this paragraph again when you can't work out why `tag-order` isn't doing what you want.

Note that the rule for testing patterns means that you can always specify a catch-all worst case by '`zstyle "*" style ...`', which will always be tried last — not just in completion, in fact, since other parts of the shell use the styles mechanism, and without the '`:completion:`' at the start of the context this style definition will be picked up there, too.

Styles like `tag-order` are the most important case where tags are used on their own. In other cases, they can be added to the end of the context; this is useful for styles which can give different results for different sets of completions, in particular styles that determine how the list of completions is displayed, or how a completion is to be inserted into the command line. The tag is the final element, so is not followed by a colon. A full context then looks something like '`:completion::complete:cd::path-directories`'. Later, you'll see some styles which can usefully be different for different tag contexts. Remember, however, that the tags part of the context, like other parts, may be empty if the completion system hasn't figured out what it should be yet.

6.5 Configuring completion using styles

You now know how to define a style for a particular context, using

```
zstyle <context> <style> <value...>
```

and some of the cases where it's useful. Before introducing other styles, here's some more detailed information. I already said that styles could take an array value, i.e. a set of values at the end of the `zstyle` command corresponding to the array elements, and you've already seen one case (`tag-order`) where that is useful. Many styles only use one value, however. There is a particularly common case, where you simply want to turn a value on or off, i.e. a boolean value. In this case, you can use any of '`true`', '`yes`', '`on`' or '`1`' for on and '`false`', '`no`', '`off`' or '`0`' for off. You define all styles the same way; only when they're used is it decided whether they should be a scalar, an array, or a boolean, nor is the name of a style checked to see if it is valid, since the shell doesn't know what styles might later be looked up. The same obviously goes for contexts.

You can list existing styles (not individually, only as a complete list) using either '`zstyle`' or '`zstyle -L`'. In the second case, they are output as the set of `zstyle` commands which would regenerate the styles currently defined. This is also useful with `grep`, since you can easily check all possible contexts for a particular style.

The most powerful way of using `zstyle` is with the option `-e`. This says that the words you supply are to be evaluated as if as arguments to `eval`. This should set the array `$reply` to the words to be used. So

```
zstyle '*' days 'Monday Tuesday'
```

and

```
zstyle -e '*' days 'reply=(Monday Tuesday)'
```

are equivalent — but the intention, of course, is that in the second case the argument can return a different value each time so that the style can vary. It will usually be evaluated in the heat of completion, hence picking up all the editing parameters; so for example

```
zstyle -e ':completion:*' mystyles 'reply=(${NUMERIC:-0})'
```

will make the style return a non-zero integer (possibly indicating `true`) if you entered a non-zero prefix argument to the command, as described in chapter 4. However, the argument can contain any zsh code whatsoever, not just a simple assignment. Remember to quote it to prevent it from being turned into something else when the `zstyle` command line is run.

Finally, you can delete a context for a style or a list of styles by

```
zstyle -d [ <context-pattern> [ <style> ] ] ...
```

— note that although the first argument is a pattern, in this case it is treated exactly, so if you give the pattern '`:completion:*:cd:*`', only values given with *exactly* that pattern will be deleted, not other values whose context begins with '`:completion:`' and contains '`:cd:`'. The pattern and the style are optional when deleting; if omitted, all styles for the context, or all styles of any sort, are deleted. The completion system has its own defaults, but these are builtin, so anything you specify takes precedence.

By the way, I did mention in passing in chapter 4 that you could use styles in just the same way in ordinary zle widgets (the ones created with '`zle -N`'), but you probably forgot about that straight away. All the instructions about defining styles and using them in your own functions from this chapter apply to zle functions. The only difference is that in that case the convention for contexts is that the context is set to '`:zle:`*widget-name*' for executing the widget *widget-name*.

The rest of this section describes some useful styles. It's up to you to experiment with contexts if you want the style's values to be different in different places, or just use '`*`' if you don't care.

6.5.1 Specifying completers and their options

'Completers' are the behind-the-scenes functions that decide what sort of completion is being done. You set what completers to use with the '`completer`' style, which takes an array of completers to try in order. For example,

```
zstyle ':completion:*' completer _complete _correct _approximate
```

specifies that first normal completion will be tried ('`_complete`'), then spelling correction ('`_correct`'), and finally approximate completion ('`_approximate`'), which is essentially the combined effect of the previous two, i.e. complete the word typed but allow for spelling mistakes. All completers set the context, so inside `_complete` you will usually find '`:completion::complete:...`', inside correction '`:completion::correct:..`', and so on.

There's a labelling feature for completers, rather like the one for tags described, but not illustrated in detail, above. You can put a completer in a list like this:

```
zstyle ':completion:*' completer ... _complete:comp-label ...
```

which calls the completer _complete, but pretends its name is comp-label when looking things up in styles, so you can try completers more than once with different features enabled. As with tags, you can write it like '_complete:-label', and the normal name will be prepended to get the name 'complete-label' — just a shortcut, it doesn't introduce anything new. I'll defer an example until you know what the completers do.

Here is a more detailed description of the existing completers; they are all functions, so you can simply copy and modify one to make your own completer.

_complete

This is the basic completion behaviour, which we've been assuming up to now. Its major use is simply to check the context — here meaning whether we are completing a normal command argument or one of the special '-context-' places — and call the appropriate completion function. It's possible to trick it by setting the parameter 'compcontext' which will be used instead of the one generated automatically; this can be useful if you write your own completion commands for special cases. If you do this, you should make the parameter local to your function.

_approximate

This does approximate completion: it's actually written as a wrapper for the _complete completer, so it does all the things that does, but it also sets up the system to allow completions with misspellings. Typically, you would want to try to complete without misspellings first, so this completer usually appears after _complete in the completers style.

The main means of control is via the max-errors style. You can set this to the maximum number of errors to allow. An error is defined as described in the manual for approximate pattern matching: a character missing such as 'rhythm' / 'rhytm', an extra character such as 'rhythm' / 'rhythms', an incorrect character such as 'rhythm' / 'rhxthm', or a pair of characters transposed such as 'rhythm' 'rhyhtm' each count as one error. Approximation will first try to find a match or matches with one error, then two errors, and so on, up to and including the value of max-errors; the set of matches with the lowest number of errors is chosen, so that even if you set max-errors large, matches with a lower number of errors will always be preferred. The real problems with setting a large max-errors are that it will be slower, and is more likely to generate matches completely unlike what you want — with typing errors, two or three are probably the most you need. Otherwise, there's always Mavis Beacon. Hence:

```
% zstyle ':completion:*' max-errors 2
# just for the sake of example...
% zstyle ':completion:*' completer _approximate
% ls
ashes    sackcloth
% echo siccl<TAB>
  -> echo sackcloth
% echo zicc<TAB>
  <Beep.>
```

because 's[i/a]c[k]cloth' is only two errors, while '[z/s][i/a]c[k]cloth' would be three, so doesn't complete.

There's another way to give a maximum number of errors, using the numeric prefix specified with ESC-<digit> in Emacs mode, directly with number keys in vi command mode, or with universal-argument. To enable this, you have to include the string numeric as one of the values for max-errors — hence this can actually be an array, e.g.

```
zstyle ':completion:*:approximate:*' max-errors 2 numeric
```

allows up to two errors automatically, but you can specify a higher maximum by giving a prefix to the completion command. So to continue the example above, enter the new zstyle and:

```
% echo zicc<ESC-3><TAB>
  -> echo sackcloth
```

because we've allowed three errors. You can start to see the problems with allowing too many errors: if you had the file 'zucchini', that would be only one error away, and would be found and inserted before 'sackcloth' was even considered.

Note that the context is examined straightaway in the completer, so at this stage it is simply ':completion::approximate:::'; no more detailed contextual information is available, so it is not possible to specify different max-errors for different commands or tags.

The final possibility as a value for the style is 'not-numeric': that means if any numeric prefix is given, approximation will not be done at all. In the last example, completion would have to find a file beginning 'zicc'.

Other minor styles also control approximation. The style original, if true means the original value is always treated as a possible completion, even if it doesn't match anything and even if nothing else matched. Completing the original and the corrections use different tags, unimaginatively called original and corrections, so you can organise this with the tag-order style.

Because the completions in this case usually don't match what's already on the command line, and may well not match each other, menu completion is entered straight away for you to pick a completion. You can arrange that this doesn't happen if there is an unambiguous piece at the start to insert first by setting the boolean style insert-unambiguous.

Those last two styles (original and insert-unambiguous) are looked up quite early on, when the context for generating corrections is being set up, so that only the context up to the completer name is available. The completer name will be followed by a hyphen and the number of errors currently being accepted. So for trying approximation with one error the context is ':completion::approximate-1:::'; if that fails and the system needs to look for completion with two errors, the context will be ':completion::approximate-2:::', and so on; the same happens with correction and 'correct-1', etc., for the completer described next.

_correct

This is very similar to _approximate, except that the context is ':completion::correct:*' (or ':completion::correct-<num>:*' when

generating corrections, as described immediately above) and it won't perform completion, just spelling correction, so extra characters which the completer has to add at the end of the word on the line now count as extra errors instead of completing in the ordinary way: `zicc` is woefully far from `sackcloth`, seven errors, but `ziccloth` only counts three again. The `_correct` completer is controlled in just the same way as `_approximate`.

There is a separate command which only does correction and nothing else, usually bound to '`^Xc`', so if you are happy using that you don't need to include `_correct` in the list of completers. If you do include it, and you also have `_approximate`, `_correct` should come earlier; `_approximate` is bound to generate all the matches `_correct` does, and probably more. Like other separate completion commands, it has its own context, here beginning '`:completion:correct-word:`', so it's easy to make this command behave differently from the normal completers.

Old-timers will remember that there is another form of spelling correction built into the shell, called with '`ESC-$`' or '`ESC-s`'. This only corrects filenames and doesn't understand anything about the new completion mechanism; the only reason for using it is that it may well be faster. However, if you use the `CORRECT` or `CORRECT_ALL` shell options, you will be using the old filename correction mechanism; it's not yet possible to alter this.

`_expand`

This actually performs expansion, not completion; the difference was explained at the start of the chapter. If you use it, you should bind tab to `complete-word`, not `expand-or-complete`, since otherwise expansion will be performed before the completion mechanism is started up. As expansion should still usually be attempted before completion, this completer should appear before `_complete` and its relatives in the list of values for the `completers` style.

The reason for using this completer instead of normal expansion is that you can control which expansions are performed using styles in the '`:completion:*:expand:*`' context. Here are the relevant styles:

`glob` expands glob expressions, in other words does filename generation using wildcards.

`substitute` expands expressions including and active '`$`' or backquotes.

But remember that you need

```
bindkey '^i' complete-word
```

when using this completer as otherwise the built-in expansion mechanism which is run by the normal binding `expand-or-complete` will take over.

You can also control how expansions are inserted. The tags for adding expansions are `original` (presumably self-explanatory), `all-expansions`, which refers to adding a single string containing all the possible expansions (the default, just like the editor function `expand-word`), and `expansions`, which refers to the results added one by one. By changing the order in which the tags are tried, as described for the `tag-order` style above, you can decide how this happens. For example,

```
zstyle ':completion:*' completer _expand _complete
zstyle ':completion::expand:*' tag-order expansions
```

sets up for performing glob expansion via completion, with the expansions being presented one by one (usually via menu completion, since there is no common prefix). Altering `expansions` to `all-expansions` would insert the list, as done by the normal expansion mechanism, while altering it to '`expansions original`' would keep the one-at-a-time entry but also present the original string as a possibility. You can even have all three, i.e. the entire list as a single string becomes just one of the set of possibilities.

There is also a `sort` style, which determines whether the expansions generated will be sorted in the way completions usually are, or left just as the shell produced them from the expansion (for example, expansion of an array parameter would produce the elements in order). If it is `true`, they will always be sorted, if `false` or unset never, and if it is `menu` they will be sorted for the `expansions` tag, but not for the `all-expansions` tag which will be a single string of the values in the original order.

There is a slight problem when you try just to generate `glob` expansions, without `substitute`. In fact, it doesn't take much thought to see that an expression like '`$PWD/*.c`' doesn't mean anything if `substitute` is inactive; it must be active to make sense of such expressions. However, this is annoying if there are no matches: you end up being offered a completion with the expanded `$PWD`, but '`*.c`' still tacked on the end, which isn't what you want. If you use `_expand` mainly for globbing, you might therefore want to set the style `subst-globs-only` to true: if a completion just expands the parameters, and globbing does nothing, then the expansion is rejected and the line left untouched.

The `_expand` completer will also use the styles

accept-exact applies to words beginning with a '`$`' or '`~`'. Suppose there is a parameter '`$foo`' and a parameter '`$foobar`' and you have '`$foo`' on the line. Normally the completion system will perform completion at this point. However, with `accept-exact` set, '`$foo`' will be expanded since it matches a parameter.

add-space means add a space after the expansion, as with a successful completion — although directories are given a '`/`' instead. For finer control, it can be set to the word `file`, which means the space is only added if the expanded word matches a file that already exists (the idea being that, if it doesn't, you may want to complete further). Both `true` and `file` may be combined with `subst`, which prevents the adding of a space after expanding a substitution of the form '`${...}`' or '`$(...)`'.

keep_prefix also addresses the question of whether a '`~`' or '`$`' should be expanded. If set, the prefix will be retained, so expanding '`~/f*`' to '`~/foo`' doesn't turn the '`~`' into '`/home/pws`'. The default is the value '`changed`', which is a half-way house been `false` and `true`: it means that if there was no other change in the word, i.e. no other possible expansion was found, the '`~`' or '`$`' will be expanded. If the effect of this style is that the expansion is the same as the unexpanded word, the next completer in the list after `_expand` will be tried.

suffix is similar to `keep_prefix`. The 'suffix' referred to is something after an expression beginning '`~`' or '`$`' that wouldn't be part of that expansion. If this style is set, and such a suffix exists, the expansion is not performed. So, for example, '`~pw<TAB>`' can be expanded to '`~pws`', but '`~pw/`' is not eligible for expansion; likewise '`$fo`' and '`$fo/`'. This style defaults to `true` — so if you want `_expand` always to expand such expressions, you will need to set it to `false` yourself.

An easier way of getting the sort of control over expansion which the `_expand` completer provides is with the `_expand_word` function, usually bound to `\C-xe`, which does all the things described above without getting mixed up with the other completers. In this case the

context string starts ':completion:expand-word', so you can have different styles for this than for the _expand completer.

Setting different priorities for expansion is one good use for completer labels, for example

```
zstyle ':completion:*' completer _expand:-glob _expand:-subst
zstyle ':completion:*:expand-glob:*' glob yes
zstyle ':completion:*:expand-subst:*' substitute yes
```

is the basic set up to make _expand try glob completions and failing that do substitutions, presenting the results as an expansion. You would almost certainly want to add details to help this along.

_history

This completes words from the shell's history, in other words everything you typed or had completed or expanded on previous lines. There are three styles that affect it, sort and remove-all-dups; they are described for the command widget _history_complete_word below. That widget essentially performs the work of this completer as a special keystroke.

_prefix

Strictly, this completer doesn't do completion itself, and should hence be in the group below starting with _match. However, it *seems* to do completion... Let me explain.

Many shells including zsh have the facility to complete only the word before the cursor, which zsh completion jargon refers to as the 'prefix'. I explained this above when I talked about expand-or-complete-prefix; when you use that instead of the normal completion functions, the word as it's finally completed looks like '<prefix><completion><suffix>' where the completion has changed '<prefix>' to '<prefix><completion>', ignoring <suffix> throughout.

The _prefix completer lets you do this as part of normal completion. What happens is that the completers are evaluated as normal, from left to right, until a completion is found. If _prefix is reached, completion is then attempted just on the prefix. So if your completers are '_complete _prefix', the shell will first try completion on the whole word, prefix and suffix, then just on the prefix. Only the first 'real' completer (_complete, _approximate, _correct, _expand, _history) is used.

You can try prefix completion more than once simply by including _prefix more than once in the list of completers; the second time, it will try the second 'real' completer in the list; so if they are '_complete _prefix _correct _prefix', you will get first ordinary completion, then the same for the prefix only, then ordinary correction, then the same for the prefix only. You can move either of the _prefix completers to the point in the sequence where you want the prefix-only version to be tried.

The _prefix completer will re-look up the completer style. This means that you can use a non-default set of completers for use just with _prefix. Here, as described in the manual, is how to force _prefix only to be used as a last resort, and only with normal completion:

```
zstyle ':completion:::::' completer _complete \
  <other-completers> _prefix
```

```
zstyle ':completion::prefix:::' completer _complete
```

The full contexts are shown, just to emphasise the form; as always, you can use wildcards if you don't care. In a case like this, you can use *only* `_prefix` as the completer, and completion including the suffix would never be tried; you then have to make sure you have the `completer` style for the `prefix` context, however, or no completion at all will be done.

The completer labelling trick is again useful here: you can call `_prefix` more than once, wherever you choose in your list of completers, and force it to look up in a different context each time.

```
zstyle ':completion:*' completer _complete _prefix:-complete \
  _approximate _prefix:-approximate
zstyle ':completion:*:prefix-complete:*' completer _complete
zstyle ':completion:*:prefix-approximate:*' completer _approximate
```

This tries ordinary completion, then the same for the prefix only, then approximation, then the same for the prefix only. As mentioned in the previous paragraph, it is perfectly legitimate to leave out the raw `_complete` and `_approximate` completers and just use the forms with the `_prefix` prefix.

One gotcha with the `_prefix` completer: you have to make sure the option `COMPLETE_IN_WORD` is set. That may sound counter-intuitive: after all, `_prefix` forces completion *not* to complete inside a word. The point is that without that option, completion is only ever tried at the end of the word, so when you type `<TAB>` in the middle of `<prefix><suffix>`, the cursor is moved to after the end of the suffix before the completion system has a chance to see what's there, and hence the whole thing is regarded as a prefix, with no suffix.

There's one more style used with `_prefix`: '`add-space`'. This makes `_prefix` add a real, live space when it completes the prefix, instead of just pretending there was one there, hence separating the completed word from the original suffix; otherwise it would simply leave the resulting word all joined together, as `expand-or-complete-prefix` usually does.

`_ignored`

Like `_prefix` this is a bit of a hybrid, mopping up after completions which have already been generated. It allows you to have completions which have already been rejected by the style '`ignored-patterns`'. I'll describe that below, but it's effect is very simple: for the context given, the list of patterns you specify are matched against possible completions, and any that match are removed from the list. The `_ignored` completer allows you to retrieve those removed completions later in your completer list, in case nothing else matched.

This is used by the `$fignore` mechanism — a list of suffixes of files not normally to be completed — which is actually built on top of `ignored-patterns`, so if you use that in the way familiar to current zsh users, where the ignored matches are shown if there are no unignored matches, you need the `_ignored` completer in your completer list.

One slightly annoying feature with `_ignored` is if there is only a single possible completion, since it will then be unconditionally inserted. Hardly a surprise, but it can be annoying if you really don't want that choice. There is a style `single-ignored` which you can set to `show` — just show the single ignored match, don't insert it — or to `menu` — go to menu completion so that TAB cycles you between the completion which `_ignored` produced and what you

originally typed. The latter gives a very natural way of handling ignored files; it's sort of saying 'well, I found this but you might not like it, so hit tab again if you want to go back to what you had before'.

I said this was like `_prefix`, and indeed you can specify which completers are called for the `_ignored` completer in just the same way, by giving the `completer` style in the context '`:completion:*:ignored:*`'. That means my description has been a little over-simplified: `_ignored` doesn't really use the completions which were ignored before; rather, when it's called it generates a list of possibilities where the choices matched by `ignore-patterns` — or internally using `$fignore` — are not ignored. So it should really be called '`_not_ignored`', but it isn't.

`_match`

This and the remaining completers are utilities, which affect the main completers given above when put into the completion list rather than doing completion themselves.

The `_match` completer should appear *after* `_complete`; it is a more flexible form of the `GLOB_COMPLETE` option. In other words, if `_complete` didn't succeed, it will try to match the word on the line as a pattern, not just a fixed string, against the possible completions. To make it work like normal completion, it usually acts as if a '`*`' was inserted at the cursor position, even if the word already contains wildcards.

You can control the addition of '`*`' with the '`match-original`' style; the normal behaviour occurs if this is unset. If it is set to '`only`', the '`*`' is not inserted, and if it is '`true`', or actually any other string, it will try first without the '`*`', then with. For example, consider typing '`setopt c*ect<TAB>`' with the `_match` completer in use. Normally this will produce two possibilities, '`correct`' and '`correctall`'. After setting the style,

```
zstyle ':completion::match:*' original only
```

no '`*`' would be inserted at the place where you hit '`TAB`', so that '`correct`' is the only possible match.

The `_match` completer uses the style `insert-unambiguous` in just the same way as does `_approximate`.

`_all_matches`

This has a similar effect to performing expansion instead of completion: all the possible completions are inserted onto the command line. However, it uses the results of ordinary contextual completion to achieve this. The normal way that the completion system achieves this is by influencing the behaviour of any subsequent completers which are called — hence you will need to put `_all_matches` in the list of completers before any which you would like to have this behaviour.

You're unlikely to want to do this with every type of completion, so there are two ways of limiting its effect. First, there is the `avoid-completer` style: you can set this to a list of completers which should *not* insert all matches, and they will be handled normally.

Then there is the style `old-matches`. This forces `_all_matches` to use an existing list of matches, if it exists, rather than what would be generated this time round. You can set the

style to `only` instead of true; in this case `_all_matches` will never apply to the completions which would be generated this time round, it will only use whatever list of completions already exists.

This can be a nuisance if applied to normal completion generation — the usual list would never be generated, since `_all_matches` would just insert the non-existent list from last time — so the manual recommends two other ways of using the completer with this style. First, you can add a condition to the use of the style:

```
zstyle -e ':completion:*' old-matches 'reply=(${NUMERIC:-false})'
```

This returns false unless there is a non-zero numeric argument; if you type `<ESC>1` in emacs mode, or just `1` in vi mode, before completion, it will insert all the values generated by the immediately preceding completion.

Otherwise, you can bind `_all_matches` separately. This is probably the more useful; copying the manual entry:

```
zle -C all-matches complete-word _generic
bindkey '^Xa' all-matches
zstyle ':completion:all-matches:*' completer _all_matches
zstyle ':completion:all-matches:*' old-matches only
```

Here we generate ourselves a new completion based on the `complete-word` widget, called `all-matches` — this name is arbitrary but convenient. We bind that to the keystroke `^Xa`, and give it two special styles which normal completion won't see. For the `completer` we set just `_all_matches`, and for `old-matches` we set `only`; the effect is that `^Xa` will only ever have the effect of inserting all the completions which were generated by the last completion, whatever that was — it does not have to be an ordinary contextual completion, it may be the result of any completion widget.

`_list`

If you have this in the list of completers (at the beginning is as good as anything), then the first time you try completion, you only get a list; nothing changes, not even a common prefix is inserted. The second time, completion continues as normal. This is like typing `^D`, then tab, but using just the one key. This differs from the usual `AUTO_LIST` behaviour in that is entirely irrespective of whether the completion is ambiguous; you always get the list the first time, and it always does completion in the usual way the second time.

The `_list` completer also uses the `condition` style, which works a bit like the styles for the `_expand` completer: it must be set to one of the values corresponding to 'true' for the `_list` delaying behaviour to take effect. You can test for a particular value of `$NUMERIC` or any other condition by using the `-e` option of `zstyle` when defining the style.

Finally, the boolean style `word` is also relevant. If false or unset, `_list` examines the whole line when deciding if it has changed, and hence completion should be delayed until the next keypress. If true, it just examines the current word. Note that `_list` has no knowledge of what happens between those completion calls; looking at the command line is its only resource.

_menu

This just implements menu completion in shell code; it should come before the 'real' completion generators in the `completers` style. It ignores the `MENU_COMPLETION` option and other related options and the normal menu-completion widgets don't work well with it. However, you can copy it and write your own completers.

_oldlist

This completer is most useful when you are in the habit of using special completion functions, i.e. commands other than the standard completion system. It is able to hang onto an old completion list which would otherwise be replaced with a newly generated one. There are two aspects to this.

First, listing. Suppose you try to complete something from the shell history, using the command bound to '`ESC-/`'. For example, I typed '`echo ma<ESC-/>`' and got '`max-errors`'. At this point you might want to list the possible completions. Unfortunately, if you type `^D`, it will simply list all the usual contextual completions — for the `echo` command, which is not handled specially, these are simply files. So it doesn't work. By putting the `_oldlist` completer into the `completers` style *before* `_complete`, it does work, because the old list of matches is kept for `^D` to use.

In this case, you can force old-listing on or off by setting the `old-list` style to `always` or `never`; usually it shows the listing for the current set of completions if that isn't already displayed, and otherwise generates the standard listing. You can even set the value of `old-list` to a list of completers which will always have their list kept in this way.

The other place where `_oldlist` is useful is in menu completion, where exactly the same problem occurs: if you generate a menu from a special command, then try to cycle through by hitting tab, completion will look for normal contextual matches instead. There's a way round this time — use the special command key repeatedly instead of tab. This is rather tedious with multiple key sequences. Again, `_oldlist` cures this, and again you can control the behaviour with a style, `old-menu`, which takes a boolean value (it is on by default). As Orwell put it, oldlisters unbellyfeel menucomp.

Ordering completers

I've given various suggestions about the order in which completers should come in, which might be confusing. Here, therefore, is a suggested order; just miss out any completers you don't want to use:

```
_all_matches _list _oldlist _menu _expand _complete _match
  _ignored _correct _approximate _prefix
```

Other orders are certainly possible and maybe even useful: for example, the `_all_matches` completer applies to all the completers following not listed in the `avoid-completer` style, so you might have good reason to shift it further down the list.

Here's my example of labels for completers, which I mentioned just above the list of different completers, whereby completers can be looked up under different names.

```
zstyle ':completion:*' completer _complete _approximate:-one \
```

```
  _complete:-extended _approximate:-four
zstyle ':completion:*:approximate-one:*' max-errors 1
zstyle ':completion:*:complete-extended:*' \
  matcher 'r:|[.,_-]=* r:|=*'
zstyle ':completion:*:approximate-four:*' max-errors 4
```

This tries the following in order.

1. Ordinary, no-frills completion.
2. Approximation with one error, as given by the second style.
3. Ordinary completion with extended completion turned on, as given by the third style. Sorry, this will be a black box until I talk about the `matcher` style later on; for now, you'll just have to take my word for it that this style allows the characters in the square brackets to have a wildcard in front, so '`a-b`' can complete to '`able-baker`', and so on.
4. Approximation with up to four errors, as given by the final style.

Here's a rather bogus example. You have a directory containing:

```
foobar  fortified-badger  frightfully-barbaric
```

Actually, it's not bogus at all, since I just created one. First try '`echo foo<TAB>`'; no surprise, you get '`foobar`'. Now try completing with '`fo-b<TAB>`' after the '`echo`': basic completion fails, it gets to '_approximate:-one' and finds that it's allowed one error, so accepts the completion '`foobar`' again. Now try '`fort-ba<TAB>`'. This time nothing kicks in until the third completion, which effectively allows it to match '`fort*-ba*<TAB>`', so you see '`fortified-badger`' (no, I've never seen one myself, but they're nocturnal, you know). Finally, try '`fortfully-ba<TAB>`'; the last entry, which allows up to four errors, thoughtfully corrects '`or`' to '`righ`', and you get '`frightfully-barbaric`'. All right, the example is somewhat unhinged, but I think you can see the features are useful. If it makes you feel better, it took me four or five attempts to get the styles right for this.

6.5.2 Changing the format of listings: groups etc.

`format`

You can use this style if you want to find out where the completions in a completion listing come from. The most basic use is to set it for the `descriptions` tag in any completion context. It takes a string value in which '`%d`' should appear; this will be replaced by a description of whatever is being completed. For example, I use:

```
zstyle ':completion:*:descriptions' format 'Completing %d'
```

and if I type `cd^D`, I see a listing like this (until I define the `group-name` style, that is):

```
Completing external command
Completing builtin command
Completing shell function
```

```
cd              cddbsubmit      cdp             cdrecord
cdctrl          cdecl           cdparanoia      cdswap
cdda2wav        cdmatch         cdparanoia-yaf
cddaslave       cdmatch.newer   cdplay
cddbslave       cdot            cdplayer_applet
```

The descriptions at the top are related to the tag names — usually there's a unique correspondence — but are in a more readable form; to get the tag names, you need to use `^Xh`. You will no doubt see something different, but the point is that the completions listed are a mixture of external commands (e.g. `cdplay`), builtin commands (`cd`) and shell functions (`cdmatch`, which happens to be a leftover from old-style completion, showing you how often I clean out my function directory), and it's often quite handy to know what you have.

You can use some prompt escapes in the description, specifically those that turn on or off standout mode ('`%S`', '`%s`'), bold text ('`%B`', '`%b`'), and underlined text ('`%U`', '`%u`'), to make the descriptions stand out from the completion lists.

You can set this for some other tag than `descriptions` and the format thus defined will be used only for completions of that tag.

group-name, group-order

In the `format` example just above, you may have wondered if it is possible to make the different types of completion appear separately, together with the description. You can do this using *groups*. They are also related to tags, although as you can define group names via the `group-name` style it is possible to give different names for completion in any context. However, to start off with it is easiest to give the value of the style an empty string, which means that group names are just the names of the tags. In other words,

```
zstyle ':completion:*' group-name ''
```

assigns a different group name for each tag. Later, you can fine-tune this with more specific patterns, if you decide you want various tags to have the same group name. If no group name is defined, the group used is called '`-default-`', so this is what was happening before you issued the `zstyle` command above; all matches were in that group.

The reason for groups is this: matches in the same group are shown together, matches in different groups are shown separately. So the completion list from the previous example, with both the `format` and `group-name` styles set, becomes:

```
Completing external command
cdctrl          cddbsubmit      cdparanoia      cdrecord
cdda2wav        cdecl           cdparanoia-yaf
cddaslave       cdot            cdplay
cddbslave       cdp             cdplayer_applet
Completing builtin command
cd
Completing shell function
cdmatch                 cdmatch.newer           cdswap
```

which you may find more helpful, or you may find messier, depending on deep psychological factors outside my control.

If (and only if) you are using `group-name`, you can also use `group-order`. As its name suggests, it determines the order in which the different completion groups are displayed. It's a little like `tag-order`, which I described when tags were first introduced: the value is just a set of names of groups, in the order you want to see them. The example from the manual is relevant to the listing I just showed:

```
zstyle ':completion:*:-command-' group-order \
    builtins functions commands
```

— remember that the '`-command-`' context is used when the names of commands, rather than their arguments, are being completed. Not surprisingly, that listing now becomes:

```
Completing builtin command
cd
Completing shell function
cdmatch                 cdmatch.newer           cdswap
Completing external command
cdctrl            cddbsubmit        cdparanoia        cdrecord
cdda2wav          cdecl             cdparanoia-yaf
cddaslave         cdot              cdplay
cddbslave         cdp               cdplayer_applet
```

and if you investigate the tags available by using `^Xh`, you'll see that there are others such as aliases whose order we haven't defined. These appear after the ones for which you have defined the order and in some order decided by the function which generated the matches.

tag-order

As I already said, I've already described this, but it's here again for completeness.

verbose, auto-description

These are relatives of `format` as they add helpful messages to the listing. If `verbose` is true, the function generating the matches may, at its discretion, decide to show more information about them. The most common case is when describing options; the standard function `_describe` that handles descriptions for a whole lot of options tests the `verbose` style and will print information about the options it is completing.

You can also set the string style `auto-description`; it too is useful for options, in the case that they don't have a special description, but they do have a single following argument, which completion already knows about. Then the description of the argument for verbose printing will be available as '`%d`' in `auto-describe`, so that something like the manual recommendation '`specify: %d`' will document the option itself. So if a command takes '`-o <output-file>`' and the argument has the description '`output file`', the '`-o`', when it appears as a possible completion, will have the description '`specify: output file`' if it does not have its own description. In fact, most options recognized by the standard completion functions already have their own descriptions supplied, and this is more subtlety than most people will probably need.

list-colors

This is used to display lists of matches for files in different colours depending on the file type. It is based on the syntax of the `$LS_COLORS` environment variable, used by the GNU version of `ls`. You will need a terminal which is capable of displaying colour such as a colour xterm, and should make sure the `zsh/complist` library is loaded, (it should be automatically if you are using menu selection set up with the `menu` style, or if you use this style). But you can make sure explicitly:

```
zmodload -i zsh/complist
```

The `-i` keeps it quiet if the module was already loaded. To install a standard set of default colours, you can use:

```
zstyle ':completion:*' list-colors ''
```

— note the use of the '`default`' tag — since a null string sets the value to the default.

If that's not good enough for you, here are some more detailed instructions. The parameter `$ZLS_COLORS` is the lowest-level part of the system used by `zsh/complist`. There is a simple builtin default, while having the style set to the empty string is equivalent to:

```
ZLS_COLORS="no=00:fi=00:di=01;34:ln=01;36:\
pi=40;33:so=01;35:bd=40;33;01:cd=40;33;01:\
ex=01;32:lc=\e[:rm=m:tc=00:sp=00:ma=07:hi=00:du=00
```

It has essentially the same format as `$LS_COLORS`, and indeed you can get a more useful set of values by using the `dircolors` command which comes with `ls`:

```
ZLS_COLORS="no=00:fi=00:di=01;34:ln=01;36:\
pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:\
or=40;31;01:ex=01;32:*.tar=01;31:*.tgz=01;31:\
*.arj=01;31:*.taz=01;31:*.lzh=01;31:*.zip=01;31:\
*.z=01;31:*.Z=01;31:*.gz=01;31:*.deb=01;31:\
*.jpg=01;35:*.gif=01;35:*.bmp=01;35:*.ppm=01;35:\
*.tga=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:\
*.mpg=01;37:*.avi=01;37:*.gl=01;37:*.dl=01;37:"
```

You should see the manual for the `zsh/complist` module for details, but note in particular the addition of the type '`ma`', which specifies how the current match in menu selection is displayed. The default for that is to use standout mode — the same effect as the sequence `%S` in a prompt, which you can display with '`print -P %Sfoo`'.

However, you need to define the style directly, since the completion always uses that to set `$ZLS_COLORS`; otherwise it doesn't know whether the value it has found has come from the user or is a previous value taken from some style. That takes this format:

```
zstyle ':completion:*' list-colors "no=00" "fi=00" ...
```

You can use an already defined `$LS_COLORS`:

```
zstyle ':completion:*' list-colors ${(s.:.)LS_COLORS}
```

(which splits the parameter to an array on colons) as `$LS_COLORS` is still useful for `ls`, even though it's not worth setting `$ZLS_COLORS` directly. This should mean GNU ls and zsh produce similar-looking lists.

There are some special effects allowed. You can use patterns to tell how filenames are matched: that's part of the default behaviour, in fact, for example '*.tar=01;31' forces tar files to be coloured red. In that case, you are limited to '`*`' followed by a string. However, there's a way of specifying colouring for any match, not just files, and for any pattern: use `=<pat>=<col>`. Here are two ways of getting jobs coloured red in process listings for the '`kill`' command.

```
zstyle ':completion:*:*:kill:*' list-colors '=%*=01;31'
```

This uses the method just described; jobs begin with '`%`'.

```
zstyle ':completion:*:*:kill:*:jobs' list-colors 'no=01;31'
```

This uses the tag, rather than the pattern, to match the jobs lines. It has various advantages. Because you are using the tag, it's much easier to alter this for all commands using jobs, not just kill — just miss out '`kill`' from the string. That wasn't practical with the other method because it would have matched too many other things you didn't want. You're not dependent on using a particular pattern, either. And finally, if you try it with a '`format`' description you'll see that that gets the colour, too, since it matched the correct tag. Note the use of the '`no`' to specify that this is to apply for a normal match; the other two-letter codes for file types aren't useful here.

However, there is one even more special effect you can use with the general pattern form. By turning on 'backreferences' with '`(#b)`' inside the pattern, parentheses are active and the bits they match can be coloured separately. You do this by extending the list of colours, each code preceded by an '=' sign, and the extra elements will be used to colour what the parenthesis matched. Here's another example for '`kill`', which turns the process number red, but leaves the rest alone.

```
zstyle ':completion:*:*:kill:*:processes' list-colors \
  '=(#b) #([0-9]#)*=0=01;31'
```

The hieroglyphics are extended globbing patterns. You should note that the `EXTENDED_GLOB` option is always on inside styles — it's required for the '`#b`' to take effect. In particular, '`#`' means 'zero or more repetitions of the previous bit of the pattern' with extended glob patterns; see the globbing manual page for full details.

ignored-patterns

Many shells, including zsh, have a parameter `$fignore`, which gives a list of suffixes; filenames ending in any of these are not to be used in completion. A typical value is:

```
fignore=(.o \~ .dvi)
```

so that normal file completion will not produce object files, EMACS backup files, or TeX DVI files.

The `ignored-patterns` style is an extension of this. It takes an array value, like `fignore`, but with various differences. Firstly, these values are patterns which should match the *whole* value to be completed, including prefixes (such as the directory part of a filename) as well as suffixes. Secondly, they apply to *all* completions, not just files, since you can use the style mechanism to tune it to apply wherever you want, down to particular tags.

Hence you can replace the use of `$fignore` above with the following:

```
zstyle ':completion:*:files' ignored-patterns '*?.o' '*?~' '*?.dvi'
```

for completion contexts where the tag '`files`' is in use. The extra '`?`'s are because `$fignore` was careful only to apply to real suffixes, i.e. strings which had something in front of them, and the '`?`' forces there to be at least one character present.

Actually, this isn't quite the same as `$fignore`, since there are other file tags than `files`; apart from those for directories, which you've already met, there are `globbed-files` and `all-files`. The former is for cases where a pattern is specified by the completion function, for example '`*.dvi`' for files following the command name `dvips`. These don't use this style, because the pattern was already sufficiently specified. This follows the behaviour for `$fignore` in the old completion system. Another slight difference, as I said above when discussing the `_ignored` completer, is that you get to choose whether you want to see those ignored files if the normal completions fail, by having `_ignored` in the completer list or not.

The other tag, `all-files`, applies when a `globbed-files` tag failed, and says any old file is good enough in that case; you can arrange how this happens with the `tag-order` style. In this example,

```
zstyle ':completion:*:*:dvips:argument*' \
  tag-order globbed-files all-files
```

is enough to say that you want to see all files if no files were produced from the pattern, i.e. if there were no '`*.dvi`' files in the directory. Finally the point of this ramble: as the `all-files` tag is separate from the `files` tag, in this case you really would see all files (except for those beginning with a '.', as usual). You might find this useful, but you can easily make the `all-files` tag behave the same way as `files`:

```
zstyle ':completion:*:(all-|)files' ignored-patterns ...
```

Here's the example of using tag labels I promised earlier; it's simply taken from the manual. To refresh your memory: tag labels are a way of saying that tags should be looked up under a different name. Here we'll do:

```
zstyle ':completion:*:*:-command-:*' tag-order 'functions:-non-comp'
```

This applies in command position, from the special '`-command-`' context, the place where functions occur most often, along with other types of command which have their own tags. This says that when functions are first looked up, they are to be looked up with the name '`functions-non-comp`' — remember that with a hyphen as the first character of the label part, the bit after the colon, the `functions` tag name itself, the bit before the colon, is to be stuck in front to give the full label name '`functions-non-comp`'. We can use it as follows:

```
zstyle ':completion:*:functions-non-comp' ignored-patterns '_*'
```

In the context of this tag label, we have told completion to ignore any patterns — i.e. any function names — beginning with an underscore. What happens is this: when we try completion in command position, `tag-order` is looked up and finds we want to try functions first, but under the name `functions-non-comp`; this completes functions apart from ones beginning with an underscore (presumably completion functions you don't want to run interactively). Since `tag-order` normally tries all the other tags, unless it was told not to, in this case all the normal command completions will appear, including functions under their normal tag name, so this just acts as a sort of filter for the first attempt at completion. This is typically what tag labels are intended for — though maybe you can think up a lot of other uses, since the idea is quite powerful, being backed up by the style mechanism.

You way wonder why you would want to ignore such functions at this point. After all, you're only likely to be doing completion when you've already typed the first character, which either is '_' or it isn't. It becomes useful with correction and approximation — particularly since many completion functions are similar to the names of the commands for which they handle completion. You don't want to be offered '`_zmodload`' as a completion if you really want '`zmodload`'. The combination of labels and ignored patterns does this for you.

You can generalise this using another feature: tags can actually be patterns, which I mentioned but didn't demonstrate. Here's a more sophisticated version of the previous example, adapted from the manual:

```
zstyle ':completion:*:*:-command-:*' tag-order \
'functions:-non-comp:non-completion\ functions *' functions
```

It's enhanced so that completion tries all other possible tags at the same time as the labelled `functions`. However, it only ever tries a tag once at each step, so the '`*`' doesn't put back `functions` as you might expect — that's still tried under the label '`functions-non-comp`', and the `ignored-patterns` style we set will still work. In the final word, we try all possible functions, so that those beginning with an underscore will be restored.

Use of the '`_ignored`' completer can allow you to play tricks without having to label your tags:

```
zstyle ':completion:*' completer _complete _ignored
zstyle ':completion:*:functions' ignored-patterns '_*'
```

Now anywhere the `functions` tag is valid, functions matching '`_*`' aren't shown until completion reaches the '`_ignored`' in the completer list. Of course, you should manipulate the completer list the way you want; this just shows the bare bones.

prefix-hidden, prefix-needed

You will know that when the shell lists matches for files, the directory part is removed. The boolean style `prefix-hidden` extends this idea to various other types of matches. The prefixes referred to are not just any old common prefix to matches, but only some places defined in the completion system: the - prefix to options, the '`%`' prefix to jobs, the - or + prefix to directory stack entries are the most commonly used.

The `prefix-needed` applies not to listings, but instead to what the user types on the

command line. It says that matches will only be generated if the user has typed the prefix common to them. It applies on broadly the same occasions as `prefix-hidden`.

`list-packed`, `list-rows-first`, `accept-exact`, `last-prompt`, `menu`

The first two of these have already been introduced, and correspond to the `LIST_PACKED` and `LIST_ROWS_FIRST` options. The `accept-exact` and `last-prompt` styles correspond essentially to the `REC_EXACT` and `ALWAYS_LAST_PROMPT` options in the same way.

The style `menu` roughly corresponds to the `MENU_COMPLETE` option, but there is also the business of deciding whether to use menu selection, as described above. These two uses don't interfere with each other — except that, as I explained, menu completion must be started to use menu selection — so a value like '`true select=6`' is valid; it turns on menu completion for the context, and also activates menu selection if there are at least 6 choices.

There are some other, slightly more obscure, choices for `menu`:

`yes=`*`num`* turn on menu completion only if there are at least *num* matches;

`no=`*`num`* turn off menu completion if there are as many as *num* matches;

`yes=long` turn on menu completion if the list does not fit on the screen, and completion was attempted;

`yes=long-list` the same, but do it even if listing, not completion, was attempted;

`select=long` like `yes=long`, but this time turn on menu selection, too;

`select=long-list` like `yes=long-list`, but turn on menu selection, too.

In case your eyes glazed over before the end, here's a full description of the last one, `select=long-list`, which is quite useful: if you are attempting completion or even just listing completions, and the list of matches would be too long to fit on the screen, then menu selection is turned on, so that you can use the cursor keys (and other selection keys) to move up and down the list. Generally, the above possibilities can be combined, unless the combined effect wouldn't work.

As always, `yes` and `true` are equivalent, as are `no` and `false`. It just hurts the eyes of programmers to read something which appears to assign a value to `true`.

`hidden`

This is a little obscure for most users. Its context should be restricted to specific tags; any corresponding matches will not be shown in completion listings, but will be available for inserting into the command line. If its value is '`true`', then the description for the tag may still appear; if the value is '`all`', even that is suppressed. If you don't want the completions even to be available for insertion, use the `tag-order` style.

6.5.3 Styles affecting particular completions

The styles listed here are for use only with certain completions as noted. I have not included the styles used by particular completers, which are described with the completer in question

in the subsection '**Specifying completers and their options**'. I have also not described styles used only in separate widgets that do completion; the relevant information is all together in the next section.

Filenames (1): patterns: `file-patterns`

It was explained above for the `tag-order` style that when a function uses pattern matching to generate file completions, such as all `*.ps` files or all `*.gz` files, the three tags `globbed-files`, `directories` and `all-files` are tried, in that order.

The `file-patterns` style allows you to specify a pattern to override whatever would be completed, even in what would otherwise be a simple file completion with no pattern. Since this can easily get out of hand, the best way of using this style is to make sure that you specify it for a narrowly enough defined context. In particular, you probably want to restrict it to completions for a single command and for a particular one of the tags usually applying to files. As always, you can use `^Xh` to find out what the context is. It has a labelling mechanism — you can specify a tag with a pattern for use in looking up other styles. Hence '`*.o:object-files`' gives a pattern '`*.o`' and a tag name '`object-files`' by which to refer to these.

The patterns you specify are tried in order; you don't need to use `tag-order`. In fact `file-patterns` replicates its behaviour in that you can put patterns in the same word to say they should be tried together, before going on to the pattern(s) in the next word. Also, you can give a description after a second colon in the same way. Indeed, since `file-patterns` gets its hands on the tags first, any ordering defined there can't be overridden by `tag-order`.

So, for example, after

```
zstyle ':completion:*:*:foo:*:*' file-patterns \
  '*.yo:yodl-files:yodl\ files *(-/):directories'
```

the command named '`foo`' will complete files ending in '`.yo`', as well as directories. For once, you don't have to change the completer to alter what's completed: '`foo`' isn't specially handled, so it causes default completion, and that means completing files, so that `file-patterns` is active anyway.

Here's a slightly enhanced example; it shows how `file-patterns` can be used instead of `tag-order` to offer the tags in the order you want.

```
zstyle ':completion:*:*:foo:*:*' file-patterns \
  '*.yo:yodl-files:yodl\ files' '*(-/):directories:directories' \
  '^*.yo(-^/):other-files:other\ files'
```

Completion will first try to show you only '`.yo`' files, if there are any; otherwise it will show you directories, if there are any; otherwise it will show you any other files: '`^*.yo(-^/)`' is an extended glob to match any file which doesn't end in '`.yo`' and which isn't a directory and doesn't link to a directory. As always, you can cycle through the sets of possibilities using the '`_next_tag`' completion command.

Note that `file-patterns` is an exception to the general rule that styles don't determine *which* tags are called only *where* they're called, or what their behaviour is: this time, you actually get to specify the set of tags which will be used. This means it doesn't use the the

standard file tags (unless you use those names yourself, of course), just '`files`' if you don't specify one. Hence it's good style to add the tags, following colons, although it'll work without.

Another thing to watch out for is that if there is already a completion which handles a file type — for example, if we had tried to alter the effect of file completion for the '`yodl`' command instead of the fictitious '`foo`' — the results may well not be quite what you want.

Another feature is that '`%p`' in the pattern inserts the pattern which would usually be used. That means that the following is essentially the same as what file completion normally does:

```
zstyle ':completion:*' file-patterns '%p:globbed-files' \
  '*(-/):directories' '*:all-files'
```

You can turn completion for a command that usually doesn't use a pattern into one that does. Another example taken from the manual:

```
zstyle ':completion:*:*:rm:*:globbed-files' file-patterns \
  '*.o:object-files' '%p:all-files'
```

So if there are any `*.o` files around, completion for `rm` will just complete those, even if arguments to `rm` are otherwise found by default file completion (which they usually are). The `%p` will use whatever file completion normally would have; probably any file at all. You can change this, if you like; there may be files you don't ever want automatically completed after `rm`.

Remember that using explicit patterns overrides the effect of `$fignore`; this is obviously useful with `rm`, since the files you want to delete are often those you usually don't want to complete.

Filenames (2): paths: ambiguous, expand, file-sort, special-dirs, ignore-parents, list-suffixes, squeeze-slashes

Filename completion is powerful enough to complete all parts of a path at once, for example '`/h/p/z`' will complete to '`/home/pws/zsh`'. This can cause problems when the match is ambiguous; since several components of the path may well be ambiguous, how much should the completion system complete, and where should it leave the cursor? This facility is associated with all these styles affecting filenames.

With ordinary completion, the usual answer is that the completion is halted as soon as a path component matches more than one possibility, and the cursor is moved to that point, with the remainder of the string left unaltered. With menu completion, you can simply cycle through the possibilities with the cursor moved to the end as usual. If you set the style `ambiguous`, then the system will leave the cursor at the point of the first ambiguity even if menu completion is in use. Note that this is always used with the '`paths`' tag, i.e. the context ends in '`...:paths`'.

The style `expand` is similar and is also applied with the '`paths`' tag. It can include either or both of the strings `prefix` and `suffix`. Be careful when setting both — they have to be separate words, for example

```
zstyle ':completion:*' expand prefix suffix
```

Don't put quotes around '`prefix suffix`' as it won't work.

With `prefix`, `expand` tells the completion system always to expand unambiguous prefixes in a path (such as '`/u/i`' to '`/usr/in`', which matches both `/usr/include` and `/usr/info`) — even if the remainder of the string on the command line doesn't match any file. So this expansion will now happen even if you try this on '`/u/i/ALoadOfOldCodswallop`', which it otherwise wouldn't.

Including `suffix` in the value of `expand` extends path completion in another way: it allows extra unambiguous parts to be added even after the first ambiguous one. So if '`/home/p/.pr`' would match '`/home/pws/.procmailrc`' or '`/home/patricia/.procmailrc`', and nothing else, the last word would be expanded. Set up like this, you will always get the longest unambiguous match for all parts of the path.

In older versions of the completion system, `suffix` wasn't used if you had menu completion active by default, although it was if menu completion was only started by the `AUTO_MENU` option. However, in recent versions, the setting is always respected. This means that setting the `expand` style to include the value `suffix` allows menu completion to cycle through all possible completions, as if there were a '`*`' after each part of the path, so '`/u/i/k`' will offer all matches for '`/u*/i*/k*`'.

The `file-sort` style allows files to be sorted in a way other than by alphabetical order: sorting applies both to the list of files, and to the order in which menu completion presents them. The value should include one of the following: '`size`', '`links`', '`modification`' (same as '`time`', '`date`'), '`access`', '`inode`' (same as '`change`'). These pick the obvious properties for sorting: file size, number of hard links, modification time, access time, inode change time. You can also add the string '`reverse`' to the value, which reverses the order. In this case the tag is always '`files`'.

The `special-dirs` style controls completion of the special directories '`.`' and '`..`'. Given that you usually need to type an initial dot to complete anything at all beginning with one, the idea of 'completing' '`.`' is a little odd; it simply means that the directory is accepted when the completion is started on it. You can set the style to `true` to allow completion to both of the two, or to '`..`' to complete '`..`' but not '`.`'. Like `ambiguous`, this is used with the tag set to '`paths`'.

The style `ignore-parents` is used with the `files` tag, since it applies to paths, but not necessarily completion of multiple path names at once; it can be used when completing just the last element. There are two main uses, which can be combined. The first case is to include the string '`parent`' in the style. This means that when you complete after (say) `foo/../`, the string `foo` won't appear as a choice, since it already appeared in the string. Secondly, you can include '`pwd`' in the value; this means don't complete the current working directory after '`../`' — you can see the sense in that: if you wanted to complete there, you wouldn't have typed the '`..`' to get out if it.

Actually, the function performs both those tests on the directories in question even if the string '`..`' itself hasn't been typed. That might be more confusing, and you can make sure that the tests for `parent` and `pwd` are only made when you typed the '`..`' by including a '`..`' in the style's value. Finally, you can include the string '`directory`' in the values: that means the tests will only be performed when directories are being completed, while if some other sort of file, or any file, can be completed, the special behaviour doesn't occur. You may have to read that through a couple of times before deciding if you need it or not.

Next, there is `list-suffixes`. It applies when expanding out earlier parts of the filename path, not just the last part. In this case, it is possible that early parts of the path were ambiguous. Normally completion stops at the point where it finds the ambiguity, and leaves the rest of the path alone. When `list-suffixes` is set, it will list all the possible values of all ambiguous components from the point of ambiguity onward.

Lastly, there is the style `squeeze-slashes`. This is rather simpler. You probably already know that in a UNIX filename multiple slashes are treated just like a single slash (with a few minor exceptions on some systems). However, path completion usually assumes that multiple slashes mean multiple directories to be completed: '`//termc`' completes to '`/etc/termcap`' because of this rule. If you want to stick with the ordinary UNIX rule you can set `squeeze-slashes` to `true`. Then in this example only files in the root directory will be completed.

Processes: `command`, `insert-ids`

Some functions, such as `kill`, take process IDs (i.e. numbers) as arguments. These can be completed by using the `ps` command to generate the process numbers. The `command` style allows you to specify which arguments are to be passed to `ps` to generate the numbers; it is simply `eval`'d to generate the command line. For example, if you are root and want to have all processes as possible completions, you might use '`-e`', for many modern systems, or '`ax`', for older BSD-like systems. The completion system tries to find a column which is headed '`PID`' or '`pid`' (or even '`Pid`', in fact) to use for the process IDs; if it doesn't find one, it just uses the first column.

The default is not to use any arguments; most variants of `ps` will then just show you interactive processes from your current session. To show all your own processes on a modern system, you can probably use the value '`ps -u$USER`' for the style — remembering to put this in single quotes. Clearly, you need to make sure the context is narrow enough to avoid unexpectedly calling odd commands.

You can make the value begin with a hyphen, then the usual command line will put afterward and the hyphen removed. The suggested use for this is adding '`command`' or '`builtin`' to make sure the right version of a command is called.

The completion system allows you to type the name of a command, for example '`emacs`', which will be converted to a PID. Note that this is different from a job name beginning with '`%`'; in this case, any command listed by `ps`, given the setting of the `command` style, can be used. Obviously, command names can be ambiguous, unlike the process IDs themselves, so the names are usually converted immediately to PIDs; if the name could refer to more than one process, you get a menu of possible PIDs.

The style `insert-ids` allows the completion system to keep using the names rather than the PIDs. If it is set to `single`, the name will be retained until you type enough to identify a particular process. If it is set to `true` (or anything else but `menu`, actually), menu completion is delayed until you have typed a string longer than the common prefix of the PIDs. This is intended to be similar to completion's usual logic — don't do anything which gets rid of information supplied by the user — so is probably more useful in practice than it sounds.

Job control: `numbers`

Builtin functions that take process IDs usually also take job specifications, strings beginning with '`%`' and followed either by a small number or a string. The style `numbers` determines how these are completed. By default, the completion system will try to complete an unambiguous string from the name of the job. If you set `numbers` to true, it will instead complete the job number — though the listing will still show the full information — and if you set it to a number, it will only use that many words of the job name, and switch to using numbers if those are not unique. In other words, if you set it to '`1`' and you have two jobs '`vi`

foo' and 'vi bar', then they will complete as '%1' and '%2' (or maybe other numbers) since the first words are the same.

Note also that prefix-needed applies here; if it is set, you need to type the '%' to complete jobs rather than processes.

System information: users, groups, hosts etc.

There are many occasions where you complete the names of users on the system, groups on the system (not to be confused with completion groups), names of other hosts you connect to via the network, and ports, which are essentially the names of internet services available on another host such as nntp or smtp.

By default, the completion system will query the usual system files to find the names of users, groups, hosts and ports, though in the final case it will only look in the file '/etc/hosts', which often includes only a very small number of not necessarily very useful hosts. It is possible to tell the completion system always to use a specified set by setting the appropriate style — users, groups, hosts, ports — to the set of possibilities you want. This is nearly always useful with hosts, and on some systems you may find it takes an inordinate amount of time for the system to query the database for groups and users, so you may want to specify a subset containing just those you use most often.

There are also three sets of combinations: hosts-ports, hosts-ports-users and users-hosts. These are used for commands which can take both or all three arguments. Currently, the command socket uses hosts-ports, telnet uses hosts-ports-users, while the style users-hosts is used by remote login commands such as rsh and ssh, and anywhere the form 'user@host' is valid.

The last is probably the most useful, so I'll illustrate that. By setting:

```
zstyle ':completion:*' users-hosts \
  pws:foo.bar.uk peters@frond.grub.uk
```

you tell rsh and friends the possible user/host combinations. Note that for the separator you can use either ':', as usual inside the completion system, or '@', which is more natural in this particular case. If you type 'rsh -l ', a username is expected and either pws or peters will be completed. Suppose you picked pws; then for the next argument, which should be a host, the system now knows that it must be foo.bar.uk, since the username for the other host doesn't match.

If you don't need that much control, completion for all these commands will survive on just the basic 'hosts', 'users', etc. styles; it simply won't be as clever in recognising particular combinations. In fact, even if you set the combined styles, anything that doesn't match will be looked up in the corresponding basic style, so you can't lose, in principle.

The other combined styles work in exactly the same way; just set the values separated by colons or '@', it doesn't matter which.

URLs for web browsers

Completion for URLs is done by setting a parallel path somewhere on your local machine. The urls style specifies the top directory for this. For example, to complete the URL

`http://zsh.org/`, you need to make a set of subdirectories of the `path` directory `http/zsh.org/`. You can extend this for however many levels of directory you need; as you would expect, if the last object is a file rather than a directory you should create it with '`touch`' rather than '`mkdir`'. The style will always use the tag '`urls`' for this purpose, i.e. the context always matches '`:completion:*:urls`'. This is a neat way of using the ordinary filing system for doing the dirty work of turning URLs into components. Arguably the system should be able to scan your browser's bookmarks file, but currently it won't; there is, however, a tool provided with the shell distribution in `Misc/make-zsh-urls` which should be able to help — ask your system administrators about this if it isn't installed, I'm sure they'll be delighted to help.

If you only have a few URLs you want to complete, you can use one of two simpler forms for the `urls` style. First, if the value of the style contains more than one word, the values are used directly as the URLs to be completed, e.g.:

```
zstyle ':completion:*:urls' urls \
    http://www.foo.org/ ftp://ftp.bar.net
```

Alternatively, you can set the `urls` style to the name of a normal file, which contains the URLs to complete separated by white space or newlines.

Note that many modern browsers allow you to miss out an initial '`http://`', and that lots of pseudo-URLs appear in newspapers and advertisements without it. The completion system needs it, however.

There is a better way when the web pages actually happen to be hosted on a system whose directories you can access directly. Set the `local` style to an array of three strings: a hostname to be considered local (you can only give one per context), the directory corresponding to the root of the files, and the directory where a user places their own web pages, relative to their home directory. For example, if your home page is usually retrieved as `http://www.footling.com/`, and that looks for the index file (often called `index.html`) in the directory `/usr/local/www/files`, and your own web pages live under '`~/www`', then you would set

```
zstyle ':completion:*:urls' local \
    www.footling.com /usr/local/www/files www
```

and when you type '`lynx http://www.footling.com/`', all the rest will be completed automatically.

The X files

There is another use for the `path` style with the tag '`colors`': it gives the path to a file which contains a list of colour names understood by the X-windows system, usually in file named '`rgb.txt`'. This is used in such contexts as '`xsetroot -solid `', which completes the name of a colour to set your root window (wallpaper) to. It may be that the default value works on your system without your needing to set this.

6.6 Command widgets

6.6.1 _complete_help

You've already met this, usually bound to '^Xh' unless you already had that bound when completion started up (in which case you should pick your own binding and use 'bindkey'), but don't forget it, since it's by far the easiest way of finding out what context to use for setting particular styles.

6.6.2 _correct_word, _correct_filename, _expand_word

The first and last of these have been mentioned in describing the related completers: _correct_word, usually bound to ^Xc, calls the _correct completer directly to perform spelling correction on the current word, and _expand_word, usually bound to ^Xe, does the same with the _expand completer. The contexts being ':completion:complete-word' and ':completion:expand-word' respectively, so that they can be distinguished in styles from the ordinary use of the completer. If you want the same styles to be used in both contexts, but not others, you should define them for patterns beginning ':completion:complete(|-word)...'.

The middle one simply corrects filenames, regardless of the completion context. Unlike the others, it can also be called as an ordinary function: pass it an argument, and it will print out the possible corrections. It does this because it bypasses most of the usual completion system. Probably you won't often need it, but it is usually bound to '^XC' (note the capital 'C').

6.6.3 _history_complete_word

This is usually bound to '<ESC-/>' for completing back in the history, and '<ESC-,>' for completing forward — this will automatically turn on menu completion, temporarily if you don't normally have that set, to cycle through the matches. It will complete words from the history list, starting with the most recent. Hence

```
touch supercalifragilisticexpialidocious
cat sup<ESC-/>
```

will save you quite a bit of typing — although in this particular case, you can use '<ESC-.>' to insert the last word of the previous command.

Various styles are available. You can set the 'stop' style which makes it stop once before cycling past the end (or beginning) of the history list, telling you that the end was reached.

You can also set the 'list' style to force matches to be listed, the 'sort' style to sort matches in alphabetical order instead of by their age in the history list, and the 'remove-all-dups' style, which ensures that each match only occurs once in the completion list — normally consecutive identical matches are removed, but the code does not bother searching for identical matches elsewhere in the list of possibilities. Finally, the range style is supported via the _history completer, which does the work. This style restricts the number of history words to be searched for matches and is most useful if your history list is large. Setting it to a number *n* specifies that only the last *n* history words should be searched for possible

matches. Alternatively, it can be a value of the form '*max*:*slice*', in which case it will search through the last *slice* history words for matches, and only if it doesn't find any, the *slice* words before that; *max* gives an overall limit on the maximum number of words to search through.

6.6.4 _most_recent_file

This function is normally bound to '^Xm'. It simply completes the most recently modified file that matches what's on the line already. Any pattern characters in the existing string are active, so this is a cross between expansion and completion. You can also give it a numeric prefix to show the Nth most recently modified file that matches the pattern.

By the way, you can actually do the same by setting appropriate styles, without any new functions. The trick is to persuade the system to use the normal _files completer with the file-sort style. By restricting the use of the styles to the context of the widget — which is simply the _generic completer described above:

```
zstyle ':completion:(match-word|most-recent-file):*' \
    match-original both
zstyle ':completion:most-recent-file::::' completer \
    _menu _files _match
zstyle ':completion:most-recent-file:*' file-sort modification
zstyle ':completion:most-recent-file:*' file-patterns \
    '*(.):normal\ files'
zstyle ':completion:most-recent-file:*' hidden true
zstyle ':completion:most-recent-file:*:descriptions' format ''
bindkey '^Xm' most-recent-file
zle -C most-recent-file menu-complete _generic
```

It may not be obvious how this works, so here's a blow by blow account if you are interested. (It works even if you aren't interested, however.)

- The 'zle -C' defines a widget which does menu completion, and behaves like ordinary completion (that's what _generic is for) except that the context uses the name of the widget we define.
- When we invoke the widget, the system uses the completer style to decide what completions to perform. This instructs it: use menu completion, complete files, use pattern matching if the completion so far didn't work.
- First, _menu comes along; it actually does nothing more than tell the system to use menu completion.
- Then _files generates a list of files. This uses the file-sort and file-patterns styles defined for the most-recent-file context. They produce a set of files in modification time order, and include only regular files (so not directories, symlinks, device files and so on).
- If that failed, the _match style allows the word on the command line to be treated as a pattern; for example, *.c to complete the most recent C source file. This uses the match-original style; the setting tells it that it should try first without adding an extra '*' for matching (this is what we want for the case where we already have a complete pattern like *.c), and if that fails, add a * at the end and try again.

- The `hidden` style means that the matches aren't listed; all that happens is the first is inserted on the line. The setting for the `format` tag similarly simplifies the display in this case by removing verbose descriptions.
- The net result is the first step of a menu completion: insert the first matched file (the most recently modified) onto the line. This is exactly what you want. Note, however, that as we are in menu completion you can keep on hitting `^xm` and the shell will cycle through the matches, which here gives you files that are progressively less recently modified.

Omit the `file-patterns` line if you don't want the match restricted to regular files (I sometimes need the most recently modified directory, but often it's irrelevant). The whole version using styles comes from Oliver Kiddle, who recommends using `_generic` in this way any time you want to generate a widget from a specific completion such as `_files`. There is a brief section on `_generic` below.

6.6.5 _next_tags

This is a very neat way of getting round the order of tags just with a key sequence. An example is the best way of showing it; it's bound by default to the key sequence '`^Xn`'.

```
% tex ^D
Completing TeX or LaTeX file
bar.tex   foo.tex   guff.tex
```

Our file is not in that directory, but by default we don't get to see the directory if there was a file that matched the pattern — here '`*.tex`'. (This will actually change in 4.1, since most people don't know about `_next_tags` but do know about directories, but you can still cycle through the different sets of tags.) You can set the `tag-order` style to alter whether they appear at the same time, but `_next_tags` lets you do this very simply. Just hit `^Xn`. You're now looking at

```
Completing TeX or LaTeX file
dir1/  dir2/  dir3/
```

and if you carry on hitting `^Xn` you will get to all files, and then you will be taken back to the `.tex` files again. (Where our file actually is, is left as an exercise for the reader.)

Of course this works with any set of tags whatsover; it simply has the effect of cycling you around the tag order.

6.6.6 _bash_completions

This function provides compatibility with a set of completion bindings in bash, in which escape followed by one of the following characters causes a certain type of (non-contextual) completion: '`!`', command names; '`$`', environment variables; '`@`', host names; '`/`', filenames, and '`~`' user names. '`^X`' followed by the same characters causes the possible completion to be listed. This function decides by examining its own binding which of those it should be doing, then calls the appropriate completion function. If you want to use it for all those possible bindings, you need to issue the right statements in your `.zshrc`, since only the bindings with '`~`' are set up by default to avoid clashes. This will do it:

```
for key in '!' '$' '@' '/'; do
  bindkey "\e$key" _bash_complete-word
  bindkey "^X$key" _bash_list-choices
done
```

Unlike most widgets, which are tied to functions of the same name to minimize confusion, the function `_bash_completions` is actually called under the names of the two different widgets shown in that code so as to be able to implement both completion and listing behaviour.

6.6.7 `_read_comp`

This function, usually bound to '`^X^R`', does on-the-fly completion. When you call it, it prompts for you to enter a type of completion; usually this will be the name of a completion function with the required arguments. Thus it's not much use unless you already have some fairly in-depth knowledge of how the system is set up. For example, try it, then enter '`_files -/`', which generates directories. There is a rudimentary completion for the function names built into it.

The next time you start it up, it will produce the same type of completion. You need to give it a numeric prefix to tell it to prompt for a different sort.

6.6.8 `_generic`

Rather than being directly bound, like the others, this widget gives you a way of creating your own special completions. You define it as a widget and bind it as if it were any completion function:

```
zle -C foo complete-word _generic
bindkey '<keys>' foo
```

Now the keys bound will perform ordinary contextual completion, but any styles will be looked up with the command context '`foo`'. So you can give it its own set of completers:

```
zstyle ':completion:foo:*' completer _expand
```

and, indeed, give it special values for any style you like. To put it another way, you've now got a complete, separate copy of the completion system where the only difference is the extra word in the context.

Good example of the use of this function were given above in the descriptions of `_all_matches` and `_most_recent_file`.

6.6.9 `predict-on`, `incremental-complete-word`

These are not really complete commands at all in the strict sense, they are normal editing commands which happen to have the effect of completion. This means that they are not part of the completion system, and though they are installed with other shell functions

they will not automatically be loaded. You will therefore need an explicit 'autoload -U predict-on', etc. — remember that the '-U' prevents the functions from expanding any of your own aliases when they are read in — as well as an explicit 'bindkey' command to bind each function, and a 'zle -N' statement to tell the line editor that the function is to be regarded as an editing widget. The predict-on file, when loaded, actually defines two functions, predict-on and predict-off, both of which need to be defined and bound for them to work. So to use all of these,

```
autoload -U incremental-complete-word predict-on
zle -N incremental-complete-word
zle -N predict-on
zle -N predict-off
bindkey '^Xi' incremental-complete-word
bindkey '^Xp' predict-on
bindkey '^X^P' predict-off
```

'Prediction' is a sort of dynamic history completion. With predict-on in effect, the line editor will try to retrieve a line back in the history which matches what you type. If it does, it will show the line, extending past the current cursor position. You can then edit the line; characters which do not insert anything mostly behave as normal. If you continue to type, and what you type does not match the line which was found, the line editor will look further back for another line; if no line matches, editing is essentially as normal. Often this is flexible enough that you can leave predict-on in effect, but you can return to basic editing with predict-off.

Note that, with prediction turned on, deleting characters reverses the direction of the history search, so that you go back to previous lines, like an ordinary incremental search; unfortunately the previous line found could be one you've already half-edited, because they don't disappear from the list until you finally hit 'return' on an edited line to accept it. There's another problem with moving around the line and inserting characters somewhere else: history searching will resume as soon as you try to insert the new characters, which means everything on the right of the cursor is liable to disappear again. So in that case you need to turn prediction off explicitly. A final problem: prediction is bad with multi-line buffers.

If prediction fails with predict-on active, completion is automatically tried. The context for this looks like ':completion:predict::::'. Various styles are useful at this point: 'list' could be set to always, which will show a possible completion even if there is only one, for example. The style 'cursor' may have the values 'complete' to move to the end of the word completed, 'key' to move past the rightmost occurrence of the character just typed, allowing you just to keep typing, or anything else not to move the cursor which is the default behaviour.

The incremental-complete-word function allows you to see a list of possible completions as you type them character by character after the first. The function is quite basic; it is really just an example of using various line editor facilities, and needs some work to make a useful system. It will understand DEL to delete the previous character, return to accept, ^G to abort, TAB to complete the word as normal and ^D to list possibilities; otherwise, keys which do not insert are unlikely to have a useful effect. The completion is done behind the scenes by the standard function complete-word.

6.7 Matching control and controlling where things are inserted

The final matter before I delve into the system for writing new completion functions is matching control; the name refers in this case to how the matching between characters already typed on the command line and characters in a trial completion is performed. This can be done in two ways: by setting the `matcher-list` style, which applies to all completions, or by using an argument (`-M`) to the low-level completion functions. Mostly we will be concerned with the first. All this is best illustrated by examples, which are taken from the section '**Matching Control**' in the `zshcompwid` manual page; in the printed manual and the 'info' pages this occurs within the section '`Completion Widgets`'.

The `matcher-list` style takes an array value. The values will be tried in order from left to right. For example,

```
zstyle ':completion:*' matcher-list 'm:{a-z-}={A-Z_}' \
       'r:|[-_./]=* r:|=*'
```

tries the first specification, which is for case-insensitive completion, and if no matches are generated tries the second, which does partial word completion; I'll explain both these specifications in detail as we go along. You can make it do both forms the second time round simply by combining the values with a space, i.e. the last word on the command line becomes `'m:{a-z-}={A-Z_} r:|[-_./]=* r:|=*'`. It is also perfectly valid to have a first matcher empty, i.e. `''`; this means that completion is tried with no matching rule the first time, and will only go on to subsequent matchers in the list if that fails. This is quite a good practice as it avoids surprises.

6.7.1 Case-insensitive matching

To perform case-insensitive matching for all completions, you can set:

```
zstyle ':completion:*' matcher-list 'm:{a-z}={A-Z}'
```

The '`m:`' specifies standard matching, with the '`{a-z}`' describing what's on the command line, and the '`{A-Z}`' what's in the trial completion. The braces indicate 'correspondence classes', which are not lessons taken by email (that's a joke), but a relative of the more usual character classes like '`[a-z]`', which, as you no doubt know, would match any of the letters between `a` and `z`. In this context, with the braces, the letters are forced to match on the left and right hand side of the '`=`', so an '`a`' on the command line must match an '`A`' in the trial completion, a '`b`' must match a '`B`', and so on. Since an `a` in the command line will always match an '`a`' in the trial completion, matcher or no matcher, this means that if you type an '`a`' it will match either '`a`' or '`A`' — in other words, case-insensitively. The same goes for any other lowercase letter you type. The difference from '`m:[a-z]=[A-Z]`' is that, because ordinary character classes are unordered, *any* lowercase letter would have matched *any* uppercase letter, which isn't what you want. The rest of the shell doesn't know about correspondence classes at all.

Finally, the use of a lowercase '`m`' at the start means that the characters actually inserted onto the line are those from the trial completion — if you type '`make<TAB>`', the completion process generates file names, and `matcher-list` allows what you type to match the file

'Makefile', then you need the latter to be inserted on the command line. Use of 'M:' at the start of the matcher would keep whatever was on the line to begin with there.

If you want completely case-insensitive matching, so that typing 'MAKE<TAB>' would also potentially complete to 'Makefile' or 'makefile' (and so on), the extension is fairly obvious:

```
zstyle ':completion:*' matcher-list 'm:{a-zA-Z}={A-Za-z}'
```

because now as well as 'a' matching 'A', 'A' will match 'a' — and, of course, 'a' and 'A' each still match themselves.

More detail on the patterns: they do not, in fact, allow all the possible patterns you can use elsewhere in the shell, since that would be too complicated to implement with little extra use. Apart from character classes and correspondence classes, you can use '?' which has its usual meaning of matching one character, or literal characters, which match themselves; or the pattern for the trial completion only can be a single '*'. which matches anything. That's it, however; you can't do other things with the '*' since it's too difficult for the system to guess what characters should be covered by it.

For the same reason, the '*' must be in an *anchored* pattern, the idea behind which is shown in the next example.

6.7.2 Matching option names

I explained back in chapter 1 that zsh didn't care too much how you specified options: 'noglob' and 'NOGLOB' and 'No_Glob' and '__NO_GLOB_' are all treated the same way. Also, this is the negation of the option 'glob'. Having learnt how to match case-insensitively, we have two further challenges: how to ignore a '_' anywhere in the word, and how to ignore the NO at the beginning so that we can complete an unnegated option name after it.

Well, here's how. Since you don't want this for all completions, just for option names, I shall show it as an argument for the 'compadd' command, which gives the system the list of possible completions. The option names should then appear as the remaining arguments to the command, and the easiest way of doing that is to have the zsh/parameter module loaded, which it always is for new completion, and use the keys of the special associative array $options:

```
compadd -M 'B:|[nN][oO]= M:_= M:{A-Z}={a-z}' - ${(k)options}
```

Here, we're interested in the thing in quotes — it means exactly the same here as it would as an element of the matcher list, except that it only applies to the trial completions given after the '-'. It's in three bits, separated by spaces; as they're in the same word, all are applied one after the other regardless of any previous ones having matched.

Starting from the right, you can see that the last part matches letters case-insensitively; the capital 'M' means that, this time, the letters on the command line, not those in the trial completion are kept; this is safe because of the way options are parsed, and reduces unexpected changes.

Moving left, you can now guess 'M:_=': it means that the '_' matches nothing at all in the trial completion — in other words, it is simply ignored. The rule for matching across the '=' is that you move from left to right, pairing off characters or elements of character classes as I already described, and when you run out, you treat any missing characters as, well, missing.

The first part has an 'anchor', indicated by what lies between the ':' and the '|'. The B specifies that the case insensitive match of 'no' must occur at the start of the word on the command line (with 'b' it would be the word in the list of matches), but here it is lax enough to allow this to happen after the 'M:_=' has stripped any initial underscores away. Hence it matches no, NO, No or nO at the start of the string, and, just like the 'M:_=' part, it ignores it, since there's nothing on the right. Again, the capital 'B' at the start means keep what's on the command line: that's important in this case, since if you lost the 'no', the meaning would change completely.

So consider the combined effect when trying to complete NO_GL. The first specification allows it to match against _GL; the second allows it to match against GL; the third, against gl; and finally the usual effect of completion means that any option beginning gl may be completed. Try 'setopt NO_GL^D' and you should see something like:

```
NO_GLob            NO_GLobassign      NO_GLobdots
NO_GLobalrcs       NO_GLobcomplete    NO_GLobsubst
```

— after the bit you've typed, the form of the words reverts to whatever's in the trial completion, i.e. lowercase letters with no '_'s.

6.7.3 Partial word completion

This example shows the other sort of anchoring, on the right, and also how to use a '*' in the right hand part of a pattern. Consider:

```
zstyle ':completion:*' matcher-list 'r:|.=* r:|=*'
```

The 'r:' specifies a right-anchored match, using the characters from the trial completion rather than what's already on the command line. As the anchor is on the right this time, the pattern (between ':' and '|') is empty, and its anchor (between '|' and '=') is '.'. So this specifies that nothing — a zero length string, or a gap between characters if you want to think of it like that — when followed by a '.', matches anything at all in the trial completion.

Consequently, the second part says that nothing anchored on the right by nothing — in other words, the right hand end of the command line string — matches anything. This is what completion normally does, add anything at all at the end of the string; we've added this part to the matcher in case the cursor is in the middle of the word. It means that the right hand end will always be completed, too.

Let's see that in action. Here are the actual contents of my actual tmp directory, never mind why:

```
regframe.rpm  t.c  testpage.dvi  testpage.log  testpage.ps
```

Now I set the matcher-list style as above and type:

```
echo t.p<TAB>
```

and get

```
echo testpage.ps
```

So, apart from the normal completion at the end (`p` to `ps`), the empty string followed by a `.` was allowed to match anything, too, and I got the effect of completing both bits of the word.

You might wonder what happens when there's a file `testpage.old.ps` around, i.e. the anchor appears twice in that. With the matcher set as given above, that won't be completed; the anchor needs to be matched explicitly, not by a wildcard. If you don't like that, you can change the '`*`' after the '`=`' in the specification to '`**`'; this form allows the anchor to occur in the string being matched. You can think of '`*`' and '`**`' as taking the shortest and the longest possible matches respectively. If you use a lot of '`**`' specifications in your matches, things can get very confusing, however.

Other shells have a facility for completing inside words like this, where it goes by such names as 'enhanced' completion, although it is usually not so flexible. In the case of tcsh, not just '`.`' but also '`-`' and '`_`' have this effect. You can force this with

```
zstyle ':completion:*' matcher-list 'r:|[._-]=* r:|=*'
```

6.7.4 Substring completion

I've mentioned '`r`' and '`B`', but corresponding to '`r`' there is '`l`', which anchors on the left instead of the right, and corresponding to '`B`' there is '`E`' which matches at the end instead of the beginning; and, of course, all exist in both upper- and lowercase forms, meaning 'keep what the user typed' and 'keep what is in the list of possible matches', respectively.

Here is an example of using '`l:|=*`' to match anything at the start of the word: this is the effect of having an empty anchor, as you saw with '`r`' above, but note with '`l`', the anchor appears, logically enough, on the left of the '`|`', in the order they would appear on the command line. By combining this with the '`r`' form, you can make the completion system work when what is on the command line matches only a substring of a trial completion — i.e., has anything else on the left and on the right. Since this can potentially generate a lot of matches, it might by an idea to try it after any other matcher specifications you have. So the following tries case-insensitive completion, then partial-word completion (case-sensitively), then substring completion:

```
zstyle ':completion:*' matcher-list 'm:{a-z}={A-Z}' \
       'r:|[._-]=* r:|=*' 'l:|=* r:|=*'
```

6.7.5 Partial words with capitals

This section illustrates another feature: if you use '`||`' when specifying anchors for '`L`' or '`R`' or their lowercase variants, the pattern part for what appears on the command line, which would usually be translated into some other pattern, is treated instead as another anchor on the other side of the pattern — which isn't matched against the pattern in the word, it just has to appear. In other words, this part matches without being 'swallowed up' in the process. An example (again adapted from the manual) will make this clearer.

```
compadd -M 'r:[^A-Z0-9]||[A-Z0-9]=** r:|=*' \
        LikeTHIS LooHoo foo123 bar234
```

The four possible completions are on the second line. The second of the two matcher specifications just allows anything to match on the right, so if we are inside the word, the

remainder may be completed. The first word is where the action is; it says 'A part of the completion which has on the left something other than an upper case letter or a digit, and on the right an upper case letter or a digit, may match anything, including the anchor'. So in particular, this would allow 'LH' to complete to 'LooHoo' — and only that, since 'LikeTHIS' has an uppercase letter to the left of the 'H', which is not allowed. In other words, the chunks of word beginning with uppercase letters and digits act like the start of substrings. (If you like, remember that last sentence and the specification, and forget the rest.)

6.7.6 Final notes

To put everything together, the possible specifications are 'm:...=...', 'l:...|...=...', 'r:...|...=...', 'b:...|...=...' and 'e:...|...=...', which cause the command line to be altered to the match found, and their counterparts with an uppercase letter, which cause what's already on the command line to be left alone and the remaining characters to be inserted directly from the completion found. The '...' are patterns, which all use the same format. They can include literal characters, a '?', and character or correspondence classes, while the rightmost pattern in each type may also consist of a '*' on its own. Characters are matched from left to right; a missing character matches an empty string, '*' matches any number of characters. Specifications may be joined in a single string, in which case all parts will be applied together.

When using the `matcher-list` style, a list of different specifications can be given; in this case, they will be tried in turn until one of them generates matches, and the rest will not be used.

There's another style apart from `matcher-list`, called `matcher`. This can be set for a particular context, possibly with specific tags, and will add the given matcher specifications using exactly the same syntax as `matcher-list` for that context, except that here all specifications are used at once, even if they are given as different elements of an array. This is possibly useful because `matcher-list` is only aware of the completer, not of any more specific part of the context.

Although I won't talk about matching control after this section, there may be cases where you want to include '`compadd -M ...`' in a completion function of your own to help the user. Many of the existing completion functions provide partial word completion where it seems useful; for example, completion of zle functions allows `i-c-w` to be completed to `incremental-complete-word` in this way.

Actually, you can configure this to a considerable extent without altering a function, using styles and labelled tags. From the manual:

```
zstyle ':completion:*:*:foo:*' tag-order '*' '*:-case'
zstyle ':completion:*-case' matcher 'm:{a-z}={A-Z}'
```

In command `foo`, whatever the tags are, they are to be tried normally first (the '*' argument to `tag-order`), then under the same name with '`-case`' appended. The second style defines a matcher for any tag ending in the suffix '`-case`', which allows lowercase characters to match uppercase ones. The upshot is that completion of anything at all for the command `foo` will be tried first case-sensitively, then case-insensitively.

6.8 Tutorial

Before bamboozling you with everything there is to know about writing your own completion function, I'll give you an example of something I wrote myself recently. If you were doing this yourself, you would then just stick this function somewhere in your function search path, and next time you started the shell it would start doing its work. However, the file already exists: it's called `_perforce` and you should find it in the function search for versions 4.1.1 and above of zsh. I apologize if it's not the ideal function to start with, but it is fresh in my mind, so what I'm saying has some chance of being correct.

This section is subtitled, 'How I struggled to write a set of completions for Perforce'. Perforce is a commercial configuration management tool (as they now call revision control systems); consult `http://www.perforce.com/` for details. It's concepts aren't a million miles from CVS, the archetypal system of this kind, but it was sufficiently different that the completion functions needed rewriting from the ground up. You won't need to know anything about CVS or Perforce, because at each stage I'll explain what I'm trying to complete and why. This should give you plenty of meat for writing completions of your own. After the tutorial, the chapter goes into the individual details, which will expand on some of the things that appeared briefly in the tutorial.

What I tend to find the most complicated part of this is making sure the completion system knows the correct types of completions and their tags to be completed at once. This probably won't be your first priority when trying to write completions of your own, but if you do it right, all the stuff about selecting types and arranging them in groups that I showed above will just work. In this tutorial we arrange to use enough of the higher level functions that it will work without too much (apparent) effort. Of course, working out from scratch which those functions are isn't always that easy; hence the tutorial.

Needless to say, I will simplify grossly at a lot of points. You can see the finished product in the zsh 4.1 distribution. It even has a few comments in.

Basic structure

Like the `cvs` command and a few other of the more complicated commands you might use, Perforce is run by a single command, `p4`, followed by an argument giving the particular Perforce command, followed by an options and arguments to that command.

This dictates the basic tasks the completion functions must do:

- If we are in the first argument, complete the name of the subcommand.
- If we are in a subsequent argument, look up the name of the subcommand and call the function which handles its arguments.

This is more complicated than most commands you will write completions for. However, one useful feature of the completion system is you can do completions in a recursive fashion. So once you get to the point where you are handling arguments for a particular subcommand, you can completely forget about the first step — as if the subcommand was the command on the line.

In addition to the subcommands, there are lots of other types of object Perforce knows about: files, obviously, plus revisions of files, set of changes ('changelists') applied at once, numbers of fixes applied to files (essentially a way of tying changlists to a particular change request

for bugtracking purposes), types of file — text, binary, etc., and several others. We will break down each of these completions into its own function. That means that any time we need to complete a particular type of object, wherever it appears (and many of these objects can appear in lots of different places), we just call the same function.

Hence there are a large number of different functions:

- The main dispatcher for the command, called `_perforce` for clarity — the main command it handles is '`p4`', but the name Perforce is more familiar.
- One function for each subcommand.
- One function for each type of object Perforce knows about and we complete (we don't bother completing dates, for example).
- In some cases, in particular files, multiple functions since there are different types of file — regular files and directories completed in the normal way, files completed by asking Perforce where it has stored them, files opened for some form of change to be made to them, and so on. Each of these is completed by a different function.

This makes it impractical to put all the functions in separate files since editing them would be a nightmare. What's more, since we will always go through the dispatcher `_perforce`, we don't need to tell the shell to autoload all the other functions; it can just hook them in from the main file. The file `_perforce` therefore has the structure:

```
#compdef p4
# Main dispatcher
_perforce() {
  # ...
}

# Helper functions for the various types of object

_perforce_files() {
  # ...
}

# ...

# Dispatchers for the individual subcommands.

_perforce_cmd_help() {
  # ...
}

# Code to make sure _perforce is run when we load it
_perforce "$@"
```

That last line is probably the least obvious. It's because of the fact that zsh (unlike other shells) usually treats the file of an autoloaded function as being the body of the function. Since everything else here just defines a function, without the last line nothing would happen the first time it was run; it would define `_perforce` and all the other functions, but that was it. The last line makes sure `_perforce` gets run with all the arguments passed down. The shell is smart enough to know that the `_perforce` function we defined in the file is the one

to keep for future use, not the entire file, so from then on things are easy; we just have a complete set of ready-defined files.

In fact the various helper functions didn't even need to use the '_' convention for completion functions, since the completion system didn't see them directly. However, I've kept it for consistency.

There's one extra trick: apart from `_perforce` itself, the function definitions look like this:

```
(( $+functions[_perforce_cmd_diff] )) ||
_perforce_cmd_diff() {
  # body of function
}
```

This is to allow the user to override each function separately. The test uses the `$functions` special associative array from the `zsh/parameter` module, which the completion system loads. If the function is already defined, because the corresponding element in the `$functions` parameter is set, then we skip the definition of the function here, because the user has already defined it. So if you were to write your own `_perforce_cmd_diff` and put it into the function path, it would be used, as you no doubt intended.

6.8.1 The dispatcher

This top level is only necessary for complex commands with multiple subcommands. There are interesting titbits here, but if you just want to know how to complete a command with ordinary UNIX-style argument parsing, skip to the next section.

The main `_perforce` function has the two purposes described at the top of the previous subsection. We need to decide whether we are in the first word after the `p4` command itself. A simple way of doing that is:

```
if (( CURRENT > 2 )); then
  # Remember the subcommand name
  local cmd=${words[2]}
  # Set the context for the subcommand.
  curcontext="${curcontext%:*:*}:p4-$cmd"
  # Narrow the range of words we are looking at to exclude `p4'
  (( CURRENT-- ))
  shift words
  # Run the completion for the subcommand
  _perforce_cmd_$cmd
else
  local hline
  local -a cmdlist
  _call_program help-commands p4 help commands | while read -A hline; do
    (( ${#hline} < 2 )) && continue
    [[ $hline[1] = (#i)perforce ]] && continue
    cmdlist=($cmdlist "${hline[1]}:${hline[2,-1]}")
  done
  _describe -t p4-commands 'Perforce command' cmdlist
fi
```

This already looks a bit horrific, but it breaks down quite easily. We test the `$CURRENT`

parameter, which is a special parameter in the completion system giving the word on the command line we are on. This is the syntactic word — the completion system has already done the hard job (and that's not an overstatement, I can tell you) of deciding what makes up a word on the command line, taking into account quoting and special characters. The array of words is stored, unsurprisingly, in the array `$words`. So word 1 will be '`p4`' and word 2 the subcommand.

Hence if we are past word 2, we look at `${words[2]}` to get the subcommand, and use that to decide what to do next. The change to `$curcontext` is a bit of cleverness to make it easy for the user to defined styles for particular subcommands; refresh your mind by looking at the discussion of styles and contexts above if you need to. For example, if you are completing after '`p4 diff`', the context will look something like '`:completion::complete:p4-diff:argument-1:opened-files`' where the remainder says you are on the first argument and are complete the tag '`opened-files`', We'll see down below how we tell the system to use that tag; the '`argument-1`' is handled by the `_arguments` utility function, which takes away a lot of the load of handling options and arguments in a standard UNIX format.

Next, we pretend that the '`p4`' at the start wasn't there by removing the front of `$words` and decrementing `$CURRENT` so as to reflect its new position in `$words`. The reason for doing this is that we are going to use `_arguments` for handling the subcommand. As is only sensible, this function looks at the first element of `$words` to find the command word, and treats the rest as options or arguments to the command.

We then dispatch the right function for the command simply by constructing the name of the function on the fly. Of course it's a little neater to check the function exists first; `$+functions[_perforce_cmd_$cmd]` would come to our aid again.

However, if we're still on the second (original) word, we have to generate a list of functions to complete. We will do this by asking Perforce's help system to list them, and store the results in the array `$cmdlist`. The loop has a couple of checks to remove blank lines and the title line at the start. The remaining lines have a command and a description. We take the command, but also tack the description on after a colon — we can then show the user the description, too, as a bit of extra help.

Actually, the Perforce command that generates the list of subcommands is simply '`p4 help command`'. (That's really all you need to know; skip the rest of the paragraph if you just want the basics.) The '`_call_program help-commands`' was stuck in front for the name of configurability. Before executing the command, the system checks in the current context with the given tag `help-commands` for the style `command`. If it finds a value for that style, it will use that as the command to execute in the place of the remaining arguments. If the style it read began with `-`, then the command it was going to execute — i.e. '`p4 help commands`' is appended to the end of the command read from the style, so that the user's command can process the original command if it needs to. This is really extreme sophistication; you will rarely actually need the `command` style, but if you are writing a completion for others to use it's polite to give them a chance to intercept calls in this way.

The `_describe` command then does the work for us. The '`-t p4-commands`' gives the tag we are going to use; the convention is that tag names are plural, though there's nothing to enforce this. Then we give an overall description — this is what appears after '`Completing `' in the examples of the `format` style above; if you don't have that set, you won't see it. Finally, we give the array name — note it is the *name*, not the substituted value. This is more efficient because the shell doesn't need to extract the values until the last minute; until then it can pass around just the single word. The `_description` function knows about the '`completion:description`' syntax; reread what I said about the `verbose` style for what

the system does with the descriptions for the completion.

The `_describe` function is one level above the completion system's basic builtin command, `compadd`; it just knows about a single tag, with a little icing sugar to display verbose descriptions. Later, we'll see ways of building up alternatives where different types of completion can be completed at the same point. There are lots of ways of doing this; some of the more complicated are relegated to the detailed descriptions that follow the tutorial.

6.8.2 Subcommand completion: `_arguments`

Suppose we are now completing after '`p4 diff`'. We have altered the command line so that the function now sees the '`diff`' as the first word, as if this were the command. This makes the next step easier; the `_arguments` function won't see irrelevant words on the command line, since it is designed to handle the arguments to a simple command in the standard form '`command [ options ] arguments ...`'. Here's the simple version.

```
_perforce_cmd_diff() {
    _arguments -s : \
        '-f[diff every file]' \
        '-t[include non-text files]' \
        '(-sd -se -sr)-sa[opened files, different or missing]' \
        '(-sa -se -sr)-sd[unopened files, missing]' \
        '(-sa -sd -sr)-se[unopened files, different]' \
        '(-sa -sd -se)-sr[opened files, same as depot]' \
        '-d-[select diff option]:diff option:'\
'((b\:ignore\ blanks c\:context n\:RCS s\:summary'\
'u\:unified w\:ignore\ all\ whitespace))' \
        "*::file:_perforce_files"
}
```

I've split the argument beginning `-d` into three lines to fit, but it's just a single argument. Also, for clarity I've missed out the line with the '`$+functions`' test to see if `_perforce_cmd_diff` was already defined; I'll forget about that for now.

The function `_arguments` has been described as having 'the syntax from hell', but with the arguments already laid out in front of you it doesn't look so bad. The are three types of argument: options to `_arguments` itself, arguments saying how to handle options to the command (i.e. '`p4 diff`'), and arguments saying how to handle normal arguments to the command.

The first two are for `_arguments` itself; '`-s`' tells it that single-letter options are allowed, i.e. they can be combined as in '`-ft`'. Luckily for our purposes, that doesn't stop us having multiple word options, too. The colon on its own then says everything else is an argument relating to the command line being handled.

We then start off with some simple options; as you can probably guess straight away, the first two say that '`p4 diff -f`' passes a flag to say any file can be diff'ed (not just ones open for editing), and that '`p4 diff -t`' passes a flag to say that binary files can be diff'ed (not just text files). Note the use of square brackets for giving a description; this is handled by the `verbose` style as I mentioned for `_describe`. In fact, the list of possible options and arguments, suitably rearranged, will end up passing through `_describe`. The descriptions in square brackets are optional, as the use of square brackets might suggest; you could just have '`-f`' and '`-t`' (making it fairly obvious why the '`:`' to separate off _arguments's own

options is a good idea).

The next step in complexity is that set of functions with the list in parentheses in front. These give mutually exclusive options. In other words, if there's already a `-sa` on the command line, don't complete any of `-sd`, `-se` or `-sr`, and so on. (Remember that by default you need to type the first '–' of an option, or the system will go straight to normal arguments, which we'll come to in a moment.)

Next comes the specification for the option `-d`. All those colons indicate that this option has an argument, and the – following straight after the `-d` indicates that it has to be in the same word, i.e. follow the `-d` without a space. After the first colon comes a description for the argument. This is what you see when you try to complete the after `-d`; compare this with the expression in square brackets before, which is what you see when you try to complete the `-d` itself. Then after the second colon is an expression saying how to complete that argument.

This final part of the specification for an option with an argument can take various forms. The simplest is just a single space; this means there's nothing to complete, but the system is aware the user needs to type something for that word and can prompt with the description. The next simplest is a set of words in parentheses: here, we could have had '`(b c n s u w)`'. Instead, we've had a variant on that which gives yet another set of descriptions, namely those for the individual completions that appear after `-d`. Note various things: the parentheses are doubled and the colons and spaces within the completion options are backslashed. All of these are simply there to make it easy for `_arguments` to parse the string. The upshot of this is that in the following context:

```
p4 diff -d
```

a verbose completion using the `format` style as described above looks like:

```
Completing diff option
b  # ignore blanks
c  # context
n  # RCS
s  # summary
u  # unified
w  # ignore all whitespace
```

or similar — I have the `list-separator` style set to '#', because it looks like a comment normal shell syntax, but in your case you may get '`--`' as the separator.

(In case you were wondering why the colons needed to be quoted when it seemed you'd already got to the last argument: it's possible for options to have multiple arguments, and you can continue having sets of :*description*:*action* pairs. This means the system needs some way of distinguishing these colons from ones inside arguments. While I'm digressing, you may also have noticed that I could have written the `-s`*X* as an option with arguments, in which case you can have a bonus point.)

The final argument starts with a '`*`', which means it applies to all remaining arguments to '`p4 diff`' after the options have been processed. Most of the rest is similar to the form for options, except for the doubled colon, which indicates that `$CURRENT` and `$words` should be altered to reflect only the arguments being handled by this argument specifier — exactly what we did before calling `_perforce_cmd_diff` in the first place, in fact. As we mentioned before, this makes the next step of processing easier if happens to call `_arguments` again. (Actually it doesn't in this case.) The '`file`' then describes the arguments and the final part, `_perforce_files`, tells the system to call that function to complete a file name.

There are numerous (it sometimes seems, endless) subtleties to _arguments. I won't try to go into them in the tutorial; see the description of _arguments below for something more detailed to refer to, and if you are feeling *really* brave look at the description in the zshcompsys manual page. Even better, dig into one of the existing completion functions — something handling completion for a UNIX command is probably good, since these make heavy use of _arguments — and see how those work. Despite the complexities, I would definitely suggest using _arguments wherever possible to take away any need on your part to do processing of command line arguments.

6.8.3 Completing particular argument types

Now we'll look inside the _perforce_files function as an example of the nitty gritty of completing one particular type of argument, which might have some quite complicated internal structure. This is true in Perforce as the filename can have extra information tacked on the end: '`file#`*revision*' indicates the revision of a file, '`file@`*change*' indicates a change status, and in some cases you can get '`file@`*change1*`,`*change2*' to indicate a range of changes (likewise revisions). Furthermore, file can be specified in different ways, and the file to be completed may be limited by some kind of context information. We'll start from simple filenames and gradually add these possibilities in.

Different types of file, part 1

There are so many possibilities for files that I'm going to split up _perforce_files into individual functions handling different aspects. For example, even if we are just handling ordinary files in the way the completion system normally does, Perforce commands understand a special file name '...' which means 'every subdirectory to any depth'. (Interestingly, zsh used to have this to mean the same thing, instead of '`**`'; it was changed in zsh because as the '.'s are regular characters there's no easy way of quoting them. You didn't need to know this.)

I'm going to say we can complete both like this:

```
_alternative \
    "files:file:_path_files" \
    "subdirs:subdirectory search:_perforce_subdir_search"
```

The function _alternative is a little bit like _arguments, but thankfully much simpler. It's name gives away its purpose; every argument specifies one of a set of possible alternatives, all of which are valid at that point — so the user is offered anything which matches out of the choices, unlike _arguments, which has to decide between the various possibilities. It's a sort of glorified loop around '_describe', with _arguments's conventions on the action for generating completions (up to a point — _alternative doesn't have all the whackier ones, though it does have the ones I've been talking about so far).

Each set of possibilities consists of the name of a tag, a description, and an argument. The tag isn't present in _arguments. If you use ^xh to tell you about valid tags, you'll see _arguments has its own generic tag, argument-rest; this isn't usually all that useful, so we are going to supply more specific ones.

In the first possibility, it's the standard one for files, 'files. The function is the basic low-level one for completing files, too; it's described below, but you already know a lot about

the effect since it's the completion system's workhorse which you use it all the time without realising. Actually, it will supply its own tags, but that doesn't matter since they will silently override what we say.

The second possibility is the new one we're adding. I've therefore invented a suitable tag 'subdirs', a description, 'subdirectory search', and the name of the function I'm going to supply to do the completion. This is quite simple:

```
_perforce_subdir_search() {
   compset -P '*/'
   compadd "$@" '...'
}
```

The first line tells the completion system to ignore anything up to the last '/'. That's so we can append a '...' to any directory which already exists on the command line. The builtin `compset` does various low-level transformations of this time. Note that the `-P` is 'greedy' — it looks for the longest possible pattern match, which is the usual default in zsh and other UNIX pattern matchers.

The second line actually adds the '...' as a completion; `compadd` is the key builtin for the whole completion system. I've actually passed some on the arguments which we got to '_perforce_subdir_search' via '"$@"'. In fact, looking back it seems as if there weren't any! However, `_alternative` actually passed some behind my back — and it's a good thing, too, since it's exactly those arguments that give the tag 'subdirs' and the description 'subdirectory search'. So that extra '"$@"' is actually quite important. The buck stops here; there's nothing below `compadd`. A function of this simplest only works well when the handling of tags and contexts has already been done; but we just saw that `_alternative` did that, so as long as we always call `_perforce_subdir_search` suitably, we're in the clear.

Different types of file, part 2

Furthermore, a Perforce file specification can look like a normal UNIX file path, or it can look like:

```
//depot/dirs/moredirs/file
```

(don't get confused with paths to network resources, which also use the doubled slash or backslash on some systems, notably Cygwin). We could use `_alternative` to handle this, too, and if I was writing `_perforce` again I probably would for simplicity. However, I decided to do it just by testing for the '//' in `_perforce_files`. This means that the structure of `p4_files` so far looks like:

```
if [[ $PREFIX = //* ]]; then
  # ask Perforce for files that match
  local -a altfiles
  altfiles=(
    'depot-files:file in depot:_perforce_depot_files'
    depot-dirs:directory in depot:_perforce_depot_dirs'
  )
  # add other alternatives, such as the '...' thing
  altfiles=($altfiles
```

```
      "subdirs:subdirectory search:_perforce_subdir_search"
    )
    _alternative $altfiles
  else
    _alternative \
        "files:file:_path_files" \
        "subdirs:subdirectory search:_perforce_subdir_search"
  fi
```

where we are still to write the functions for the first two alternatives in the first branch; the '. . .' is still valid for that branch, so I've added that as the third alternative. I've used the array `$altfiles` because, actually, the structure is more complicated than I've shown; doing it this way makes it easier to add different sets of alternatives.

The choice of which branch is made by examining the `$PREFIX` special variable, which contains everything (well, everything interesting) that comes before the cursor position in the word being completed. There is a counterpart `$SUFFIX` which we will see in a moment. The 'almost everything' comes because sometimes we definitely don't want to see the whole `$PREFIX`. Completing the three dots was such as case — we didn't want to see anything up to the last `/`. What that '`compset -P '*/'`' actually did was move the matched pattern from the front of `$PREFIX` to the end of `$IPREFIX`, another special parameter which contains parts of the completion we aren't currently interested in, but which are still there. This allows us to concentrate on a particular part of the completion. However you do that — whether by `compset` or directly manipulating `$PREFIX` and friends — the completion system usually restores the parameters when you exit the function where you altered them. This fits in nicely with what we're doing here with `_alternative` — if we handle adding '. . .' by ignoring everything up to the last slash, for example, we don't want the next completion we try to continue to ignore that; other file completions will want to look at the directory path.

'Depot' is Perforce's name for what CVS calls a repository — the central location where all versions of all files are stored, and from where they are retrieved when you ask to look at one. I've separated out '`depot-dirs`' and '`depot-files`' for various reasons. First, the commands to examine files and directories are different, so the completion function is different. Second, we can offer different tags for files and directories — this is what `_path_files` does for normal UNIX files. Third, it will later allow us more control — some commands only operate on directories. Here's `_perforce_depot_files`; `_perforce_depot_dirs` is extremely similar:

```
  _perforce_depot_files() {
    # Normal completion of files in depots
    local pfx=${(Q)PREFIX} expl
    local -a files

    compset -P '*/'
    files=(${${${(f)"$(\
_call_program files p4 files \
\"\$pfx\*\$\{\(Q\)SUFFIX\}\" 2>/dev/null)"}%\#*}##*/})
    [[ $#files -eq 1 && $files[1] = '' ]] && files=()
    compadd "$@" -a files
}
```

A little messy (and still not quite the full horror). I've split the key line in the middle which fetches the list from Perforce to make it fit. If you ploughed through chapter 5, you'll recognised what's going on here — we're reading a list of files, one per line, from the command

'`p4 files`', and we're stripping off the directory at the front, and everything from a '`#`' on at the end. The latter is a revision number; we're not handling those at this point, though we will later.

Notice the way I remembered `$PREFIX` before I told the system to ignore it for the word we're now completing. I remembered it as '`${(Q)PREFIX}`' in order to remove any quotes from the name. For example, if the name on the line so far had a space, `$PREFIX` (which comes from what is on the command line without any quotes being stripped) would have the space quoted somehow, e.g. '`name\ with\ space`'. We arrange for `$pfx` to contain '`name with space`', which is how Perforce knows the file, using the `(Q)` parameter flag. We then pass the argument "`$pfx*${(Q)SUFFIX}`" to '`p4 files`'; this generates matching files internally. The extra layer of backslash-quoting is for the benefit of `_call_program`, which re-evaluates its arguments; this ensures the argument is expanded at the point it gets passed to `p4 files`. All this goes to show just how difficult getting the quoting right can be.

Once we've got the list of bare filenames, we check to see if the list is just one element with no length. That's an artefact of the the "`$(cmd)`" syntax; if the output is empty, because its quoted you still get one zero-length string output, which we don't want.

Finally, we pass the result to `compadd` as before. Again, tags and the description have already been handled and we just need to make sure the appropriate options get passed in with "`$@`". This time we use the '`-a`' option which tells `compadd` that any arguments are array name, not a list of completions. This is more efficient; compadd only needs to expand the array internally instead of the shell passing a potentially huge list to the builtin.

Handling extra bits on a completion

'Extra bits' on a completion could be anything; common examples include an extra value for a comma-separated list (the `_values` functions is for this), or some kind of modifier applied to the completion you have already. We've already seen an example, in fact, since the principle of handling the directory and basename parts of a file is very similar. The phrase 'extra bits' may already alert you to the fact that we are heading towards the deeper recesses of completion.

Anyway, here's how we tack a revision or change number onto the end of a file.

I'll stick with revisions: '*filename*`#`*revision*', where *revision* is a number. For the full sophistication, there are three steps to this. First, make it easy for the user to add '`#`' to an existing filename; second, recognise that a '`#`' is already there so that revisions need to be completed; third, find out the actual revisions which can be completed. As a revision is just a number, you might think completing it was a bit pointless. However, given the sophistication of zsh's completion system there's actually one very good reason — we can supply a description with the revisions, so that the user is given information about the revisions and can pick the right one without running some external command to find out. There was the same sort of rationale behind the '`-d`' option to `p4 diff`; there was just one letter to type, but zsh was able to generate extra information to describe the possibilities, so it wasn't just laziness.

First part: make it easy for the user to add the '`#`'. This actually depends on a new feature in version 4.1 of zsh; in 4.0 you couldn't play the trick we need or grabbing the keyboard input after a completion was finished unless you specified a particular suffix to add to the completion (such as the '`/`' after a directory — this is historically where this feature came from).

The method is to add an extra argument everywhere we complete a file name. For example, change the `compadd` in `_perforce_depot_files` to:

```
compadd "$@" -R _perforce_file_suffix -a files
```

where the option argument specifies a function:

```
  _perforce_file_suffix() {
    [[ $1 = 1 ]] || return

    if [[ $LBUFFER[-1] = ' ' ]]; then
        if [[ $KEYS = '#' ]]; then
            # Suffix removal with an added backslash
            LBUFFER="$LBUFFER[1,-2]\\"
        elif [[ $KEYS = (*[^[:print:]]*|[[:blank:]\;\&\|@]) ]]; then
            # Normal suffix removal
            LBUFFER="$LBUFFER[1,-2]"
        fi
    fi
}
```

This has been simplified, too; I've ignored revision ranges in the form *file*`#`*rev1*`,`*rev2*. However, I've handled changes ('`@`' following a filename) as well as revisions. You'll see this function looks much more like a zle widget rather than a completion widget — which is exactly what it is; it's not called as part of the completion system at all. After the specified completion, zle reads in the next keystroke, which is stored in `$KEYS`, and calls this function as a zle widget. This means it can manipulate the line buffer; we only need to look at what is at the left of the cursor, stored in `$LBUFFER`.

The function is called with the length of the suffix added to the function. In this case, it's just a space — we've finished a normal completion, so the system has automatically added a space to what's on the command line. We therefore check we've just got one single character in the suffix, to avoid getting confused.

Next, we look at what's immediately left of the cursor, which is the last character in `$LBUFFER`, i.e. `$LBUFFER[-1]`, to make sure this is a space.

If everything looks OK, we consider the keys typed and decide whether to modify the line. You may already have noticed that in some cases zsh automatically removes that space by itself; for example, if you hit return — or any other non-printing character — or if it's a character that terminates a command such as '`&`' or '`;`'. We emulate that behaviour — most of the second test is simply to do that. The only differences from normal are if the key typed was '`@`' or '`#`'.

The '`@`' is simple — we just remove the last character, the same as we do for the other characters. For '`#`', however, we also add a backslash to the command line before the '`#`'. That's because '`#`' is a special character with extended globbing, and the completion system generally runs with extended globbing switched on. Adding the backslash means the user doesn't have to; it's never harmful.

To show the next effect, suppose we complete a file name:

```
p4 diff fil<TAB>
```

to get:

```
p4 diff filename _
```

where '_' shows the cursor position, and then typed '#'; we would get:

```
p4 diff filename\#
```

with the cursor right at the end.

So far so good. For the second step, we need to modify `_perforce_files` to spot that there is a '#' on the line before the cursor, and to call the revision code. To do this we add an extra branch at the start of the '`if`' in `_perforce_files` — at the start, because any '#' before the cursor forces us to look at revisions, so this takes precedence over the other choices. When this is added, the code will look like:

```
  if [[ -prefix *\# ]]; then
    _perforce_revisions
  elif [[ $PREFIX = //* ]]; then
    # as before.
```

In fact, that `-prefix` test is just a fancy way of saying the same thing as the '`[[ $PREFIX = *\# ]]`' and if I wasn't so hopelessly inconsistent I would have written both tests the same.

So now the third step: write `_perforce_revisions` to complete revisions numbers with the all-important descriptions.

```
_perforce_revisions() {
   local rline match mbegin mend pfx
   local -a rl

   pfx=${${(Q)PREFIX}%%\#*}
   compset -P '*\#'

   # Numerical revision numbers, possibly with text.
   if [[ -z $PREFIX || $PREFIX = <-> ]]; then
       # always allowed (same as none)
       rl=($rl 0)
       _call_program filelog p4 filelog \$pfx 2>/dev/null |
          while read rline; do
           if [[ $rline = (#b)'... #'(<->)*\'(*)\' ]]; then
               rl=($l "${match[1]}:${match[2]}")
           fi
       done
   fi
   # Non-numerical (special) revision names.
   if [[ -z $PREFIX || $PREFIX != <-> ]]; then
       rl=($rl 'head:head revision' 'none:empty revision'
               'have:current synced revision')
   fi
   _describe -t revisions 'revision' rl
}
```

Thankfully, a lot of the structure of this is already familiar. We extract the existing prefix before the '#', being careful about quoting — this is the filename for which we want a list of revisions. We ignore everything in the command argument before the '#'. After generating the completions, we use the `_describe` function to add them with the tag '`revisions`' and the description '`revision`'.

The main new part is the loop over output from '`p4 filelog`', which is the Perforce command that tells us about the revisions of a file. We extract the revision number and the comment from the line using backreferences (see previous chapter) and weld them together with a colon so that `_describe` will be able to separate the completion from its description. Then we add a few special non-numerical revisions which Perforce allows, and pass this list down to `_describe`. The extra `if`'s are a very minor optimization to check if we are completing a numerical or non-numerical revision.

6.8.4 The rest

It's obvious that this tutorial could expand in any number of directions, but as it's really just to point out some possibilities and directions, that would would miss the point. So the rest of this chapter takes the completion system apart and looks at the individual components. It should at least now be a bit more obvious where each component fits.

6.9 Writing new completion functions and widgets

Now down to the nitty gritty. When I first talked about new completion, I explained that the functions beginning '_' were the core of the system. For the remainder of the chapter, I'll explain what goes in them in more detail than I did in the tutorial. However, I'll try to do it in such a way that you don't need to know every single detail. The trade off is that if you just use the simplest way of writing functions, many of the mechanisms I told you about above, particularly those involving styles and tags, won't work. For example, much of the code that helps with smart formatting of completion listings is buried in the function '`_description`'; if you don't know how to call that — which is often done indirectly — then your own completions won't appear in the same format as the pre-defined ones.

The easiest way of getting round that is to take a dual approach: read the following as far as you need, but also try to find the existing completion that comes nearest to meeting your needs, then copy that and change it. For example, here's a function that completes files ending in `.gz` (the supplied function which does this has now changed), which are files compressed by the `gzip` program, for use by the corresponding program that does decompression, `gunzip` — hence the file and function are called `_gunzip`:

```
#compdef gunzip zcat

local expl

_description files expl 'compressed file'
_files "$expl[@]" -g '*.[gG][zZ]'
```

You can probably see straight away that if you want to design your own completion function for a command which takes, say, files ending in `.exe`, you need to change three things: the line at the top, which gives the names of programmes whose arguments are to be completed

here, the description '`compressed file`' to some appropriate string, and the argument following the `-g` to something like `'*.exe'` — any globbing pattern should work, just remember to quote it, since it shouldn't be expanded until the inside of the function `_files`. Once you've installed that somewhere in your `$fpath` and restarted the shell, everything should work, probably following a longer pause than usual as the completion system has to rescan every completion function when it finds there is a new one.

What you might miss is that the first argument to `_description`, '`files`', is the all-important mystical tag for the type of completion. In this case, you would probably want to keep it. Indeed, the `_files` function is used for all file completions of any type, and knows all about the other tags — `globbed-files`, `directories`, `all-files` — so virtually all your work's done for you here.

If you're adding your own functions, you will need your own functions directory. This was described earlier in this guide, but just to remind you: all you need to do is create a directory and add it to `$fpath` in either `.zshenv` (which a lot of people use) or `.zshrc` (which some sticklers insist on, since it doesn't affect non-interactive shells):

```
fpath=(~/funcs $fpath)
```

It's best to put it before the standard completion directories, since then you can override a standard completion function simply by copying it into your own directory; that copy will then be found first and used. This is a perfectly reasonable thing to do with any completion function — although if you find you need to tweak one of the larger standard functions, that's probably better done with styles, and you should suggest this to us.

6.9.1 Loading completion functions: `compdef`

The first thing to understand is that top line of `_gunzip`. The '`#compdef`' tag is what tells the system when it checks through all files beginning with '`_`' that this is a function implementing a completion. Files which don't directly implement completions, but are needed by the system, instead have the single word '`#autoload`' at that point. All files are only loaded when needed, using the usual autoloading system, to keep memory usage down.

You can supply various options to the '`#compdef`' tag; these are listed in the '`Initialization`' section of the `zshcompsys(1)` manual page or '**Completion System**' info node. The most useful are `-k` and `-K`, which allow you to define a completion command and binding rather than a function used in a particular context. There are also `-p` and `-P` which tell the system that what follows is a pattern rather than a literal command name; any command matching the pattern will use that completion function, unless you used `-P` and a normal (non-pattern) completion function for the name was found first.

For normal `#compdef` entries, however, what comes next is a list of command names — or rather a list of contexts, since the form '`-context-`' can be used here. For example, the function `_default` has the line '`#compdef -default-`'. You can give as many words as you like and that completion will be used for each. Note that contexts in the colon-separated form can't appear here, just command names or the special contexts named with hyphens.

The system does its work by using a function `compdef`; it gets as arguments more or less what you see, except that the function name is passed as the first argument. Thus the `_gunzip` completion is loaded by '`compdef _gunzip gunzip zcat`', `_default` by '`compdef _default -default-`', and so on. This simply records the name of the function handling the context in the `$_comps` associative array which you've already met. You can

make extra commands/contexts be handled by an existing completion function in this way, too; this is generally more convenient than copying and modifying the function. Just add '`compdef <_function> <command-to-handle>`' to `.zshrc` after the call to `compinit`.

It's also high time I mentioned an easy way of using the completion already defined for an existing function: '`compdef newcmd=oldcmd`' tells the completion system that the completion arguments for '`newcmd`' are to be the same as the ones already defined for '`oldcmd`'; it will complain if nothing is known about completing for `oldcmd`. This works recursively; you can now define completions in terms of that for `newcmd`. If you happen to know the name of the completion function called, you can use that; the following three lines are broadly equivalent:

```
compdef $_comps[typeset] foo
compdef _vars_eq foo
compdef foo=typeset
```

since the completion for `typeset` is stored in `$_comps` along with all the others, and this happens to resolve to `_vars_eq`; but the last example is easier and safer and the intention more obvious. The manual refers to `typeset` here as a 'service' for `foo` (guess what the shell stores in the associative array element `$_services[foo]`).

There's actually more to services: when a function is called, the parameter `$service` is set. Usually this will just be the name of the command being completed for, or one of the special contexts like '`-math-`'. However, in a case like the last `compdef` in the list above, the service will be `typeset` even though the command name may be '`foo`'.

This is also used in '`#compdef`' lines. The top of '`_gzip`' contains:

```
#compdef gzip gunzip gzcat=gunzip
```

which says that the file provides two services, for `gzip` and `gunzip`, and also handles completion for `gzcat`, but with the service name `gunzip`. Only a few of the completion functions actually care what service they provide (you can check, obviously, by looking to see if they refer to `$service`); but you may have uses for this. Note that if you define services with a `compdef` command, *all* the arguments must be in the *foo=bar* form; the mixed form is only useful after a `#compdef` inside completion functions.

6.9.2 Adding a set of completions: `compadd`

Once you know how to make a new completion function, there is only one other basic command you need to know before you can create your own completions yourself. This is the builtin `compadd`. It is at the heart of the completions system; all its arguments, after the options, are taken as possible completions. This is the list from which the system selects the possibilities that match what you have already typed. Here's a very basic example which you can type or paste at the command line:

```
_foo() { compadd Yan Tan Tethera; }
compdef _foo foo
```

Now type '`foo `' and experiment with completions after it. If only it were all that simple.

There are a whole list of options to `compadd`, and you will have to look in the `zshcompwid(1)` manual page or the '**Completion Widgets**' info node for all of them.

I've already mentioned `-M` and (long ago) `-f`. Here are other interesting ones. `-X <description>` provides a description — this is used by the `format` style to pass descriptions, and if you use the normal tags system you shouldn't pass it directly; I'll explain this later.

`-P <prefix>` and `-S <suffix>` allow you to specify bits which are not treated as part of the completion, but appear on the line none the less. In fact, they do two different things: if the prefix or suffix is already there, it is ignored, and if it isn't, it is inserted. There are also corresponding hidden and ignored prefixes, necessary for the full power of the completion system, but you will need to read the manual for the full story. The `-q` option is useful with `-S`; it enables auto-remove behaviour for the suffix you gave, just like `/` with the `AUTO_REMOVE_SLASH` option when completing filenames.

`-J <group>` is the way group names are specified, used by the `group-name` tag; there is also `-V <group>`, but the group here is not sorted (and is distinct from any group of the same name passed to `-J`). `-Q` tells the completion code not to quote the words — this is useful where you need to have unquoted metacharacters in the final completion. It is also useful when you are completion something where the result isn't going to be expanded by the shell.

`-U` tells `compadd` to use the list of completions even if they don't match what's on the command line; you will need this if your completion function modifies the prefix or suffix so that they no longer fit what's already there. If you use this, you might consider turning on menu completion (using `compstate[insert]=menu`), since it might otherwise be difficult to select the appropriate completion.

Finally, note the `-F` and `-W` options which I describe below for `_files` actually are options to `compadd` too.

6.9.3 Functions for generating filenames, etc.

However, for most types of completion the possibilities will not be a simple list of things you already know, so that you need to have some way of generating the required values. In this section, I will describe some of the existing functions you can call to do the hard work. In the next section I will show how to retrieve information from some special parameters made available by the `zsh/parameter` module.

Files etc.: the function `_files`

You have already seen `_files` in action. Calling this with no arguments simply adds all possible files as completions, taking account of the word on the command line to establish directories and so on.

For more specific use, you can give it various options: '`-/`' means complete directories, and, as you saw, '`-g "<pattern>"`' gives a filename generation pattern to produce matching files.

A couple of other options, which can be combined with the ones above, are worthy of mention. If you use '`-W <dir>`', then completion takes place under directory `<dir>` rather than in the current directory — it has no effect if you are using an absolute path. Here, '`<dir>`' can also be a set of directories separated by spaces or, most usefully since it avoids any problems with quoting, the name of an array variable which contains the list of possible directories. This is essentially how completion for `cd` with the `$cdpath` array works. So if you have a program

that looks for files with the suffix '.mph', first in the current directory, then in a standard directory, say, /usr/local/oomph', you can do this:

```
  local oomph_dirs
  oomph_dirs=(. /usr/local/oomph)
_files -W oomph_dirs -g '*.mph'
```

— note there is no '$' before the variable $oomph_dirs here, since it should only be expanded deep inside _files.

The system that implements $fignore and the ignored-patterns style can be intercepted, if you need to, with the option '-F "<pat>"'; '<pat>' is an array of patterns to ignore, in the usual completion format, in other words the name of a real shell array, or a list of values inside parentheses. If you make sure all the tags stuff is handled properly, ignored-patterns will work automatically, however, and in addition extended globbing allows you to specify patterns with exclusion directly, so you probably won't use this feature directly unless you're in one of your superhero moods.

In addition, _files also takes many of the standard completion options which apply to compadd, for convenience.

Actually, the function _path_files is the real engine room of the system. The advantage of using _files is that it prepares all the tags for you, deciding whether you want directories to be completed as well as the globbed files, and so on. If you have particularly specific needs you can use _path_files directly, but you won't get the automatic fallback one directories and all-files. Because it doesn't handle the tags, _path_files is too lowly to do the usual tricks with label loops, i.e. pretending 'dog:-setter' is a tag 'dog-setter' with the usual completions for 'dog'; likewise, it doesn't implement the file-patterns style. So you need to know what you're doing when you use it directly.

Parameters and options

These can be completed by calls to the _parameters and _options functions, respectively. Both set up their own tags, and _options uses the matching control mechanism described above to allow options to be given in all the available forms. As with _files, they will also pass standard compadd options down to that function. Furthermore, they are all at a high enough level to handle tags with labels: to translate that into English, you can use them directly without any of the preprocessing described later on which are necessary to make sure the styles dealing with tags are respected.

For more detailed control with options, the functions _set_options and _unset_options behave like _options, but the possible completions are limited to options which are set or unset, respectively. However, it's not that simple: the completion system itself alters the options, and you need to enable some code near the top of _main_complete (it's clearly marked) to remember the options which were set or unset when completion started. A straw poll based on a sample of two zsh developers revealed that in any case many people don't like the completion system to second guess the options they want to set or unset in this way, so it's probably better just to stick to _options.

Miscellaneous

There are also many other completion functions adding matches of a certain type. These can be used in the same way as `_parameters` and `_options`; in other words they do all the work needed for tags themselves and can be given options for `compadd` as arguments. Normally, these functions are named directly after the type of matches they generate, like `_users`, `_groups`, `_hosts`, `_pids`, `_jobs`, etc.

6.9.4 The `zsh/parameter` module

The new completion system automatically makes the `zsh/parameter` module available for use. This provides an easy way of generating arguments for `compadd`. To get the maximum use out of this, you should be familiar with zsh's rather self-willed syntax for extracting bits out of associative arrays. Note in particular `${(k)assoc}`, which expands to a list of the keys of the associative array `$assoc`, `${(v)assoc}`, which expands to just its values (actually, so does `$assoc` on its own), and `${(kv)assoc}` which produces key/value pairs. For all intents and purposes, the keys and values, or the pairs of them, are in a random order, but as the completion system does it's own sorting that shouldn't be a problem. Mostly, the important parts for completion are in the keys, i.e. to add all aliases as possible completions, you need '`compadd ${(k)aliases}`'.

Here's a list of associative and ordinary arrays provided; for more information on the values of the associative arrays, which could be useful in some cases, consult the section **The zsh/parameter Module** in the `zshmodules(1)` manual page or the corresponding info node. First, the associative arrays.

`$aliases`, `$dis_aliases`, `$galiases` The keys of these arrays give ordinary aliases, disabled ordinary aliases for those where you have done `disable -a <alias>` to turn them off temporarily, and global aliases as defined with `alias -g`.

`$builtins`, `$dis_builtins` The keys give active and disabled shell builtin commands.

`$commands` The keys are all external commands stored in the shells internal tables; it does this both for the purposes of fast completion, and to avoid having to search each time a command is executed. It's possible that a command is missing or incorrectly stored if the contents of your `$path` directories has changed since the shell last updated its tables; the `rehash` command fixes it.

`$functions`, `$dis_functions` The keys are active and disabled shell functions.

`$history` Here, the *values* are complete lines stored in the internal history. The keys are the numbers of the history line; it's an associative, rather than an ordinary, array because they don't necessarily start at line 1. However, see the `historywords` ordinary array below.

`$jobtexts`, `$jobdirs`, `$jobstates` These give you information about jobs; the keys are the job numbers, as presented by the `jobs` command, and the values give you the other information from jobs: `$jobtexts` tells you what the job is executing, `$jobdirs` its working directory, and `$jobstates` its state, where the bit before the colon is the most useful as it refers to the whole job. The remainder describes the state of individual processes in the job.

`$modules` The keys give the names of modules which are currently available to the shell, i.e. loaded or to be autoloaded, essentially the same principle as with functions.

$nameddirs If you have named directories, either explicitly (e.g. assigning '`foo=/mydir`' and using '`~foo`') or via the `AUTO_NAME_DIRS` option, the keys of this associative array give the names and the values the expanded directories.

$options, $parameters The keys give shell options and parameters, and are used by the functions `_options` and `_parameters` for completion, so you will mostly not need to refer to them directly.

$userdirs The keys give all the users on the system. The values give the corresponding home directory, so '`${userdirs[juser]}`' is equivalent to having `~juser` expanded and is thus not all that interesting, except that by doing it this way you can test whether the expansion exists without causing an error.

Now here are the ordinary arrays, which you would therefore refer to simply as `${reswords}` etc.

$dirstack This contains your directory stack, what you see with '`dirs -v`'. Note, however that the current directory, which appears as number 0 with that command, doesn't appear in `dirstack`. Of course it's easy to add it to a completion if you want.

$funcstack This is the call stack of functions, i.e. all the functions which are active at the time the array was referenced. `^Xh` uses this to display which functions have been called for completion.

$historywords Unlike `$history`, this contains just the individual words of the shell's command line history, and is therefore likely to be more useful for completion purposes.

$reswords, $dis_reswords The active and disabled reserved words (effectively syntactically special commands) understood by the shell.

Other ways of getting at information

Since the arguments to `compadd` undergo all the usual shell expansions, it's easy to get words from other sources for completion, and you can look in the existing completion functions for many examples. A good understanding of zsh's parameter and command expansion mechanisms and a strong stomach will be useful here.

For example, here is the expansion used by the `_limits` function to retrieve the names of resource limits from the `limit` command itself:

```
print ${${(f)"$(limit)"}%% *}
```

which you can test does the right thing. Here's a translation: `"$(limit)"` calls the command in a quoted context, which means you get the output as if it were a single file (just type '`limit`' to see what that is). `${(f)...}` splits this into an array (it is now outside quotes, so splitting will generate an array) with one element per line. Finally, `${...%% *}` removes the trailing end of each array element from the first piece of whitespace on, so that '`cputime unlimited`' is reduced to '`cputime`', and so on. Type '`limit ^D`', and you will see the practical upshot of this.

That's by no means the most complicated example. The nested expansion facility is used throughout the completion functions, which adds to brevity but subtracts considerably from readability. It will repay further study, however.

6.9.5 Special completion parameters and `compset`

Up to now, I've assumed that at the start of your completion function you already know what to complete. In more complicated cases that won't be the case: different things may need completing in different arguments of a command, or even some part of a word may need to be handled differently from another part, or you need to look for a word following a particular option. I will first describe some of the lower level facilities which allow you to manipulate this; see the manual page `zshcompwid(1)` or the info node **Completion Widgets** for the details of these. Later, I will show how you can actually skip a lot of this for ordinary commands with options and arguments by using such functions as `_arguments`, where you simply specify what arguments and options the function takes and what sort of completion they need.

The heart of this is the special parameters made available in completion for testing what has already been typed. It doesn't matter if there are parameters of that name outside the completion system; they will be safely hidden, the special values used, and the original values restored when completion is over.

`$words` is an array corresponding to the words on the command line — where by a 'word' I mean as always a single argument to the command, which may include quoted whitespace. `$CURRENT` is the index into that array of the current word. Note that to avoid confusion the ksh-like array behaviour is explicitly turned off in `_main_complete`, so the command itself is `$words[1]`, and so on.

The word being completed is treated specially. The reason is that you may only want to complete some of it. An obvious example is a file with a path: if you are completing at '`foo/bar`', you don't want to have to check the entire file system; you want the directory `foo` to be fixed, and completion just for files in that. There are actually two parts to this. First, when completion is entered, `$PREFIX` and `$SUFFIX` give you the part of the current word before the cursor, and the remainder, respectively. It's done like this to make it possible to write functions for completing inside a word, not just at the end. The simplest possible way of completing a file is then to find everything that matches `$PREFIX*$SUFFIX`.

But there's more to it than that: you need to separate off the directory, hence the second part. The parameters `$IPREFIX` and `$ISUFFIX` contain a part of the string which will be ignored for completion. It's up to you to decide what that is, then to move the bit you want to be ignored from `$PREFIX` to `$IPREFIX` (that's the usual case) or from `$SUFFIX` to `$ISUFFIX`, making sure that the word so far typed is still given by `$IPREFIX$PREFIX$SUFFIX$ISUFFIX`. Thus in completing `foo/bar`, you would strip `foo/` from the start of `$PREFIX` and tack it onto the end of `$IPREFIX` — after recording the fact that you need to move to directory `foo`, of course. Then you generate files in `foo`, and the completion system will happily accept `barrack` or `barbarous` as completions because it doesn't care about the `foo` any more.

Actually, this is already done by the the `_files` and `_path_files` functions for filename completion. Also, you can get some help using the `compset` builtin command. In this case, the incantation is

```
if compset -P "*/"; then
  # do whatever you need to with the leading
  # string up to / stripped off
else
  # no prefix stripped, do whatever's necessary in this case
fi
```

In other words, any initial match of the pattern '*/' in $PREFIX is removed and transferred to the end of $IPREFIX; the command status tells you whether this was done. Note that it is the longest possible such match, so if there were multiple slashes, all will be moved into $IPREFIX. You can control this by putting a number <N> between the -P and the pattern, which says to move only up to the <N>th such match; here, that would be a pattern with exactly <N> slashes. Note that -P stands for prefix, not pattern; there is a corresponding -S option for the suffix. See the manual for other uses of compset; these are probably the most frequent.

If you want to make the test made by compset, but without the side effect of changing the prefixes and suffixes, there are tests like this:

```
if [[ -prefix */ ]]; then
  # same as with `compset -P "*/"', except prefixes were left alone.
fi
```

These have the advantage of looking like all the standard tests understood by the shell.

There are three other parameters special to completion. The $QIPREFIX and $QISUFFIX are a special prefix and suffix used when you are dividing up a quoted word — for example, in 'zsh -c "echo hi"', the word "echo hi" is going to be used as a command line in its own right, so if you want to do completion there, you need to have it split up. You can use 'compset -q' to split a word in this fashion.

There is also an associative array $compstate, which allows you to inspect and change the state of many internal aspects of completion, such as use of menus, context, number of matches, and so on. Again, consult the manual for more detail. Many of the standard styles work by altering elements of $compstate.

Finally, in addition to the parameters special to completion, you can examine (but not alter) any of the parameters which appear in all editing widgets: $BUFFER, the contents of the current editing line; $LBUFFER, the part of that before the cursor; $RBUFFER, the rest; $CURSOR, the index of the cursor into $BUFFER (with the first character at zero, in this case — or you can think of the zero as being the point before the first character, which is where insertion would take place with the cursor on the first character); $WIDGET and $LASTWIDGET, the names of the current and last editing or completion widget; $KEYS, the keys typed to invoke the current widget; $NUMERIC, any numeric prefix given, unset if there is none, and a few other probably less useful values. These are described in the zshzle(1) manual page and the **Zsh Line Editor** info node. In particular, I already mentioned $NUMERIC as of possible use in various styles, and it is used by the completers which understand a 'numeric' value in their relevant styles; the $WIDGET and $KEYS parameters are useful for deciding between different behaviours based on what the widget is called (as in _history_complete_word), or which keys are used to invoke it (as in _bash_completions).

Here are a few examples of using special parameters and compset.

One of the shortest standard completions is this, _precommand:

```
#compdef - nohup nice eval time rusage noglob nocorrect exec

shift words
(( CURRENT-- ))

_normal
```

It applies for all the standard commands which do nothing but evaluate their remaining arguments as a command, with some change of state, e.g. ignoring a certain signal (`nohup`) or altering the priority (`nice`). All the completion system does here is shift the first word off the end of the `$words` array, decrement the index of the current word into `$words`, and call `_normal`. This is the function called when completion occurs not in one of the special `-context-`s, in other words when an argument to an ordinary command is being completed. It will look at the new command word `$words[1]`, which was previously the first argument to `nohup` or whatever, and start completion again based on that, or even complete that word itself as a command if necessary. The net effect is that the first word is ignored completely, as required.

Here's just an edited chunk of the file `_user_at_host`; as its name suggests, it completes words of the form `<user>@<host>`, and it's used anywhere the `user-hosts` style, described above, is appropriate:

```
if [[ -prefix 1 *@ ]]; then
  local user=${PREFIX%%@*}

  compset -P 1 '*@'

  # complete the host for which we want the user
else
  # no @, so complete the user
fi
```

We test to see if there is already a '`<user>@`' part. If there is, we extract the user with an ordinary parameter substitution (so ordinary even other shells could do it). Then we strip off that from the bit to be completed with `compset`; we already know it matches the prefix, so we don't need to test the return value. Then we just do normal hostname completion on what remains — except that the `user-hosts` style might be able to give us a clue as to which hosts have such a user. If the original test failed, then we simply complete what's there as a user.

Finally, here is essentially what the function `_most_recent_file` uses to extract the `$NUMERIC`th (default first) most recently modified file.

```
local file
file=($~PREFIX*$~SUFFIX(om[${NUMERIC:-1}]N))
(( $#file )) && compadd -U -i "$IPREFIX" -I "$ISUFFIX" -f -Q - $file
```

Instead of doing it with mirrors, this uses globbing qualifiers to extract the required file; `om` specifies ordering by modification time, and the expression in square brackets selects the single match we're after. The `N` turns on `NULL_GLOB`, so `$file` is empty if there are no matches, and the parameter expansions with '`$~`' force patterns in `$PREFIX` and `$SUFFIX` to be available for expansion (a little extra feature I use, although ordinary completion would work without).

Most of the `compadd` command is bookkeeping to make sure the parts of the prefix and suffix we've already removed, if there are any, get passed on, but the reason for that deserves a mention, since normally this is handled automatically. The difference here is that `-U` usually replaces absolutely everything that was in the word before, so if you need to keep it you have to pass it back to `compadd`. For example, suppose you were in a context where you were completing after '`file=...` and you had told the completion system that everything up to '`file=`' was not to count and not to be shown as part of the completion. You would want

to keep that when the word was put back on the command line. However, '`-U`' would delete that too. Hence the '`-i "$IPREFIX"`' to make sure it's retained. The same argument goes for the ignored suffix. However, there's currently no way of getting `_most_recent_file` to work on only a part of a string, so this explanation really only applies when you call it from another completion function, not directly from the command line.

6.9.6 Fancier completion: using the tags and styles mechanism

At this point, you should be in a position to construct, although maybe not in the best possible way, pretty much any completion list you want. Now I need to explain how you make sure it all fits in with the usual tags and styles system. You will need to pick appropriate tags for your completions. Although there is no real restriction, it's probably best to pick one of the standard tags, some of which are suitably general to cover just about anything: `files`, `options`, `values`, etc. There is a list in the completion system manual entry. Remember that the main use for tags is to choose what happens when more than one tag can be completed in the same place. Finding such things that can't be separated using the standard tag names is a good reason for inventing some new ones; you don't have to do anything special if the tag names are new, just make sure they're documented for anyone using the completion function.

How to call functions so that 'It Just Works'

The simplest way of making your own completion function recognize tags is to use the `_description` function, which is usually called with three arguments: the name of the tag you're completing for, the name of a variable which will become an array containing arguments to pass to `compadd`, and the full description. Then you have to make sure that array gets passed down to `compadd`, or to any of the higher-level completion functions which will pass the arguments on to `compadd`. For example,

```
local expl
_description files expl 'my special files'
_files "$expl[@]"
```

This sets the files tag; `_description` sets `$expl` to pass on the description, and maybe other things such as a group name for the tag, in the appropriate format; we pass this down to `_files` which will use it for calling `compadd`. Generally, you will call `_description` for each time you call `compadd` or something that in turn calls `compadd`.

The `_description` function calls another function `_setup` to do much of the setting up of styles for the particular tag. Mostly, `_setup` is buried deeply enough that you don't need to worry about it yourself. Sometimes you can't do completion, and just want to print a message unconditionally to say so, irrespective of tags etc.; the function `_message` does this, taking the message as its sole argument.

There are two levels above that; these implement the tags mechanism in full. In `_description`, all that happens is that the user is informed what tag is coming up; there's no check what preferences the user has for tags (the first level), nor whether he wants tags to be split up using the labelling mechanism, e.g. picking out certain sorts of files using the labelled tag '`file:-myfiles`' to get the final tag '`file-myfiles`' (the second level).

To get this for simple cases you use the function `_wanted`. Unlike `_description`, it's an interface to the function that generates completion as well as a handler for tags — that's so

it can loop over the generated tags, checking the labels. The call above would now look like this:

```
_wanted files expl 'my special files' _files
```

Note that you now don't pass the "`$expl[@]`", which hasn't even been set yet; `_wanted` will generate the string using the parameter name you say (here '`expl`', as usual), and assume that the function generating the completions can use the result passed down to it. This is true of pretty much anything you are likely to want to use.

Note also the fact you need to pass '`_files`', i.e. the function generating the completion. You can put pretty much any command line which generates completions here, down to a simple '`compadd`' expression. The reason it has to be here is the tag labelling business: `_wanted` could check whether the tag you specify, '`files`', is wanted by the user and then return control to you, but it wouldn't be able to split up and loop over labelled tags set in this case for the `file-patterns` style and in other case by the `tag-order` style.

Unless you're really going into the bowels, `_wanted` is probably the lowest level you will want to use. I'd suggest you remember that one, and only go back and look at the other stuff if you need to do something more complicated.

If your function handles multiple tags, you need to loop over the different tags to find out which sort the tag order wants next. For this, you first need to tell the system which tags are coming up, using the `_tags` function with a list. Then you need to to test whether each tag in turn actually needs to be completed, and go on doing this until you run out of tags which need completions performing; the `_tags` function without arguments does this. Finally, you need to use `_requested`, which works a bit like `_wanted` but is made to fit inside the loop we are using. The end result looks like this:

```
local expl ret=1
_tags foo bar rod
while _tags; do
  _requested foo expl "This is the description for tag foo" \
    compadd all foos completions && ret=0
  _requested bar expl "This is the description for tag bar" \
    compadd all bars completions && ret=0
  _requested rod expl "This is the description for tag rod" \
    compadd all rods completions && ret=0
  (( ret )) || return 0   # leave if matches were generated
done
```

If you do include the completion function line as arguments, the loop over labels for the tag you specify is automatically handled as with `_wanted`. It may be a little confusing that both `_requested` and `_wanted` exist: the specific difference is that with `_requested` you call the `_tags` function yourself, whereas `_wanted` assumes the only valid tag is its argument and acts accordingly, and can be used only for simple, 'one-shot' completions.

With `_requested`, unlike `_wanted`, you can separate out the arguments to the completion generator itself — here `compadd` — into a different statement, remembering the "`$expl[@]`" argument in that case. You can miss out the second and third arguments for `_requested` in this way. This time the loop which generates labels for tags is not performed, and you have to arrange it yourself, with the usual trade off of greater complexity for greater flexibility. To do this, there are two other functions: `_all_labels` and `_next_label`. The

simpler case is with `_all_labels`, which just implements the loop over the labels using the same arguments as `_wanted`:

```
_requested values &&
   _all_labels values expl 'values for my special things'  \
     compadd alpha bravo charlie delta echo foxtrot.
```

In case you haven't understood (and it's quite complicated, I'm afraid): the `_requested` looks at whether the tag you use has been asked for by the user. Having found out that it is, the `_all_labels` function calls the command `compadd` which actually adds the completions, but it does it in such a way as to take account of labelled tags — you might have both a plain '`values`' tag and '`values:-special`' labelled tag, and `_all_labels` is needed to decide which is being used here. This last example is actually exactly what `_requested` does when given the `compadd` as argument, so it's only really useful when there is some code between the `_requested` and the `_all_labels`, for example to compute the strings to complete.

The most complicated case you are likely to come across is when inside the part of the tags loop which handles a particular tag (i.e. the `_requested` lines in the example above), you actually want to add more than one possible sort of completion. Then `_all_labels` is no longer enough, because completion needs to sort out the different things which are being added. This can also happen when there is only one valid tag, but that has multiple completions so that `_wanted` isn't any use. In this case you need to use `_next_label` inside a loop, which, as its names suggests, fixes up labels for the current tag and stops when it's found the right one. Here's a stripped down example which handles completion of messages from the `MH` mail handling system; you'll find it complete inside the function `_mh`.

```
_tags sequences
while _tags; do
  while _next_label sequences expl sequence; do
    compadd "$expl[@]" $(mark $foldnam 2>/dev/null |
                         awk -F: '{ print $1 }') && ret=0
    compadd "$expl[@]" reply next cur prev \
        first last all unseen && ret=0
    _files "$expl[@]" -W folddir -g '<->' && ret=0
  done
  (( ret )) || return 0
done
```

Here's what's going on. The `_tags` call works just as it did in the first example I showed for that, deciding whether the tag in question, `sequences`, has been asked for; the tag name comes because MH allows you to define sets of messages called exactly 'sequences'. The first '`while`' selects all values from `tag-order` where the '`sequences`' tag appears, with or without a label. The second '`while`' loop then sorts out any occurrences of labelled sequences to be presented to the user at the same time, i.e. given in the same element of the `tag-order` value array. The first `compadd` extracts from the folder (MH's name for a directory) identified by the function the names of any sequences you have defined; the second adds a lot of standard sequences — although strictly speaking `unseen` isn't a standard sequence since you can name it yourself in `~/.mh_profile`. Finally, the third adds files in the folder itself whose names are just digits, which is how MH stores messages. The handling of `return` makes sure it stops as soon as you have matches for one particular element of `tag-order`; if you put it in the inner loop, you would just have the first of those sets that happened to be generated, while here, if you specify that all types of sequence should appear in the same completion list, they are all correctly collected.

Why, in that last example, is there no call to _requested, now I've gone to the trouble of explaining what that does? The answer is that there is only one tag; _tags can decide if we want it at all, and after that the tag is known, so we don't need _requested to find that information out for us. It's only needed if there is more than one type of match — indeed, that's why we introduced it, so this is not actually a new complication, although you can be forgiven for thinking otherwise.

Here's an example of using that code for sequences. You might decide that you only want to see named sequences unless there aren't any, otherwise ordinary messages. You could do this by setting your styles as follows:

```
zstyle ':completion:*' tag-order sequences:-name sequences:-num
zstyle ':completion:*:sequences-name' ignored-patterns '(|,)<->'
zstyle ':completion:*:sequences-num' ignored-patterns  '^<->'
```

which tries sequences under the labels sequences-name and sequences-num; which ignore completions which are all digits, and those which are not all digits, respectively. The slight twiddle in the pattern for sequences-name ignores messages marked for deletion as well, which have a comma stuck in front of the number (this is configurable, so your version of MH may be different).

All of _description, _wanted, _requested, _all_labels and _next_label take the options -J and -V to specify sorted or unsorted listings and menus, and the options -1 and -2 for removing consecutive duplicates or all duplicates. These are also options to compadd; the reason for handling them here is that they can be different for each tag, and the function called will set expl appropriately.

If your requirements are simple enough, you can replace that _tags loop above with a single function, _alternative. This takes a series of arguments each in the form '<tag>:<description>:<action>', with the first two in the form you now know, and the third an action. These are essentially the same as actions for the _arguments function, described below, except that the form '->state', which says that the calling function will handle the action itself by using the value of the parameter $state, is not available. The most common forms of action here will be a call to another completion function, maybe with arguments (e.g. '_files -/'), or a simple list in parentheses (e.g. '(see saw margery daw)'). Here, for example, is how the _cd function handles the two cases of local directories (under the current directory) and directories reached via the $cdpath parameter:

```
local tmpcdpath
tmpcdpath=(${(@)cdpath:#.})
_alternative \
    'local-directories:local directories:_path_files -/' \
    'path-directories:directories in cdpath:
_path_files -W tmpcdpath -/'
```

The only tricky bit is that $tmpcdpath: it removes the '.' from $cdpath, if it's present, so that the current directory is always searched for with the tag 'local-directories', never with 'path-directories'. Actually, you could argue that it should be treated as being in 'path-directories' when it's present; but that confuses the issue over what 'local-directories' really means, and it is useful to have the distinction.

It's now an easy exercise to replace the example function I gave for _requested by a call to _alternative with the arguments to compadd turned into a list in parentheses as the <action> part of the arguments to _alternative.

How to look up styles

If your completion function gets really sophisticated, you may want it to look up styles to decide what its behaviour should be. The same advice goes as for tags: only invent a new style if the old ones don't seem to cover the use you want to make, since by using contexts you can always restrict the scope of the style. However, by the same token don't try to squeeze too much meaning into one style, which will force the user to narrow the context — it's always much easier to set a style for the general context '`:completion:*`' than to have to worry about all the circumstances where you need a particular value.

Retrieving values of styles is no harder than defining them, but you will need to know about the parameter `$curcontext`, which is what stores the middle part of the context, sans '`:completion:`' and sans tag. When you need to look something up, you pass this context to `zstyle` with '`:completion:`' stuck in front:

```
zstyle -b ":completion:${curcontext}:tag" style-name parameter
```

If the tag is irrelevant, you can leave it empty, but you still need the final colon since there should always be six in total. In some cases where multiple tags apply it's useful to have a `:default` tag context as a fall back if none of the actual tags yield styles for that context; hence you should test the style first for the specific tag, then with the `default`.

Style lookups all have the form just shown; the result for looking up `style-name` in the given context will be saved in the `parameter` (which you should make local, obviously). In addition, `zstyle` returns a zero status if the lookup succeeded and non-zero if it failed. The `-t` lookup is different from the rest as it only returns a status for a boolean, i.e. returns status 0 if the value is `true`, `yes`, `1` or `on`, and doesn't require a parameter name. There is also a `-T`, which is identical except that it returns status 0 if the style doesn't exist, i.e. the style is taken to default to true.

The other lookup options return the style as a particular type in the parameter with exit status zero if the lookup succeeded, i.e. a value was found, and non-zero otherwise; `-b`, `-s`, and `-a` specify boolean (`parameter` is either `yes` or `no`), scalar (`parameter` is a scalar), and array (`parameter` is an array, which may still be a single word, of course), You can retrieve an associative array with `-a` as long as the parameter has already been declared as one.

There's also a convenience option for matching, `-m`; instead of a `parameter` this takes a `pattern` as the final argument, and returns status zero if and only if the `pattern` matches one of the values stored in the style for the given context.

Typical usages are thus:

```
if zstyle -t ":completion:${curcontext}:" foo; then
  # do things in a fooish way
else
  # do things in an unfooish way
fi
```

or to use the value:

```
local val
if zstyle -s ":completion:${curcontext}:" foo val; then
  # use $val to establish how fooish to be
```

```
else
  # be defaultly fooish
fi
```

6.9.7 Getting the work done for you: handling arguments etc.

The last piece of unfinished completion business is to explain the higher level functions which can save you time writing completions for commands which behave in a standard way, with arguments and options. The good news is that all the higher functions here handle tags and labels internally, so you don't need to worry about `_tags`, `_wanted`, `_requested`, etc. There's one exception: the 'state' mechanism to be described, where a function signals you that you're in a given state using the parameter `$state`, expects you to handle tag labels yourself — pretty reasonable, as you have requested that the function return control to you to generate the completions. I've mentioned that here so that I don't have to gum up the description of the functions in this section by mentioning it again.

Handling ordinary arguments

The most useful function is `_arguments`. There are many examples of this in the completion functions for external commands, since so many external commands take the standard format of a command with options, some taking their own arguments, plus command arguments.

The basic usage is to call it with a series of arguments (which I'll call 'specifications') like:

```
<where I am>:<description>:<what action to take>
```

although there are a whole series of more complicated possibilities.

The initial '`<where I am>`' part tells the function whether the specification applies to an argument in a particular position, or to an option and possibly any arguments for that option. Let's start with ordinary arguments, since these are simpler. In this case '`<where I am>`' will be either a number, giving the number of the argument, or a '`*`', saying that this applies to all remaining arguments (or all arguments, if you haven't used any of the other form). You can simplify the first form, by just missing out the number; then the function will assume it applies to the first argument not yet specified. Hence the standard way of handling arguments is with a series of specifications just beginning '`:`' for arguments that need to be handled their own way, if any, then one beginning '`*:`' for all remaining arguments, if any.

The message that follows is a description to be passed on down to `_description`. You don't specify the tags at this point; that comes with the action.

The action can have various forms, chosen to be easily distinguishable from one another.

1. A list of strings in parentheses, such as '`(red blue green)`'. These are the possible completions, passed straight down to `compadd`.

2. The same, but with double parentheses; the list in this case consists of the completion, a backslashed colon, and a description. So an extended version of the previous action is '`((red\:The\ colour\ red blue\:The\ colour\ blue))`' and so on. You can escape other colons inside the specifications in this way, too.

3. A completion function to call, with any arguments, such as '`_files -/`' to complete directories. Usually this does the business with `$expl` which should be familiar from the section on basic tag handling, however you can put an extra space in front of the action to have it called exactly as is, after word splitting.

4. A word preceded by '`->`' for example '`->state`'. This specifies that `_arguments` should return and allow the calling function to process the argument. To signal back to the calling function, the parameter `$state` will be set to what follows the '`->`'. It's up to the calling function to make `$state` a local parameter — `_arguments` can't do that, since then it couldn't return a value.

   You should also make the parameters `$context` and `$line` local; the former is set to the new part to be added to `$curcontext`, which, as you can find out from `^Xh`, is `option-<option>-<arg>`, for example `option-file-1` for the first argument of the `option-file` option, or `argument-N`, for example `argument-2` for the second argument of the command.

   In simple cases, you will just test the parameter `$state` after `_arguments` has returned to see what to do: the return value is 300 to distinguish it from other returns where `_arguments` itself performed the completion.

5. A chunk of code to evaluate, given in braces, which removes the need for a special function or processing states. Obviously this is best used for the simplest cases.

These are the main possibilities, but I have not described every variation. As always, you should see the manual for all the detail.

Here's a concocted example for that '`->state`' action specifier, in case it's confusing you. It's for a command that takes arguments '`alpha`', '`beta`' and '`gamma`', and takes a single option '`-type`' which takes one argument, either '`normal`' or '`unusual`'.

```
local context state line
typeset -A opt_args

_arguments '-type[specify type]:type:->type' \
           '*:greek letter:->gklet' && return 0

case $state in
  (type)  compadd normal unusual && return 0
          ;;
  (gklet) compadd alpha beta gamma && return 0
          ;;
esac

return 1
```

In fact the possibilities here are so simple that you don't need to use `$state`; you can just use the form with the values in parentheses as the action passed to '`_arguments`'. Anyway, if you put this into a function '`_foo`', type '`compdef _foo foo`', and attempt completion for the fictitious command '`foo`', you will see `_arguments` in action.

I haven't shown the gory tag handling; as it's written, you'll see that no tag is ever defined for the `compadd` arguments shown. In this case you could just use `_wanted`. What you get for free with arguments, however, is the context: in the first case, you would have '`:option-type-1`' in the argument field (the second last, just before the tag), and in the

second case ':argument-rest:'. Go back to where I originally described contexts if you've forgotten about these; I didn't tell you at the time, but it's the _argument function that is responsible for them. (However, you can supply a '-C' argument to _wanted to tell that a context.)

A note about the form: that '&& return 0' makes the completion function return if _arguments was satisfied that it found a completion on its own. It's useful in more complex cases. Remember that most completion functions return status zero if and only if matches were added; this function is written to follow that convention. I already showed this in the section on tags, but you might have skipped that.

Note all the things you had to make local: $context, $state, $line and the associative array $opt_args. The last named allows you to retrieve the values for a particular option; for example '$opt_args[-o]' contains any value already on the command line for the option -o. For options that take multiple arguments, these appear separated by colons, so if the line contains '-P prefix 3', $opt_args[-P] will contain 'prefix:3'.

Handling options

Option handling is broadly similar, with the '<where I am>' part just giving the option name — I already showed one example with '-type' above. In this case, the option will just be completed to itself, the first part of the specification, and the rest says how to complete its arguments. Since options can take any number of arguments, including zero, the :description:action pair can be repeated, or omitted entirely. Otherwise, it behaves similarly to the way described for ordinary command arguments, with all the same possible actions. So a simple option specification could be

```
_arguments '-turnmeon'
```

for an option with no arguments,

```
_arguments '-file:input file:_files'
```

for an option with one argument, or

```
_arguments '-iofiles:input file:_files:output file:_files'
```

for an option with two arguments, both files but with different descriptions.

The first part of the specification for an option can be more complicated, to reflect the fact that options can be used in all sorts of different ways. You can specify a description for the option itself — as I tried to explain, the descriptions in the rest of the specification are instead for the arguments to the option. To specify an option description, just put that after the option, before any colons, in square brackets:

```
_arguments '-on[turn me on, why not]'
```

Next, some options to a command are mutually exclusive. As _arguments has to read its way along the command line to parse it, it can record what options have already appeared, and can ensure that an option incompatible with one there already will not be completed. To do this, you need to include the excluded option in parentheses before the option itself:

```
_arguments '(-off)-on[turn me on, why not]' \
           '(-on)-off[turn me off, please]'
```

This completes either of the options '-on' or '-off', but if you've already given one, it won't complete the other on the same command line. If you need to give multiple excluded options, just list them separated by spaces, like '(-off -noton)'.

Some options can themselves be repeated; _arguments usually won't do that (in a sense, they are mutually exclusive with themselves), but you can allow it to happen by putting a '*' in front of the option specification:

```
_arguments '*-o[specify extra options]:option string:->option'
```

allows you to complete any number of '-o <option>' sets using the $state mechanism. The * appears after any list of excluded options.

There are also ways of allowing different methods of option handling. If the option is followed by -, that means the value must be in the same word as the option, instead of in the next word; if that is allowed, but the argument could be in the next word instead, the option should be followed by a '+'. The latter behaviour is very common for commands which take single letter options. Some commands, particularly many recent GNU commands, allow you to have the argument in the next word or in the current word after an '=' sign; you get this by putting an '=' after the option name. For example,

```
_arguments '-file=:input file:_files'
```

allows you to complete '-file *<filename>*' or '-file=*<filename>*'. With

```
 _arguments '-file=-:input file:_files'
```

only the second is possible, i.e. the argument must be after the '=', not in its own word.

You can handle optional and repeated arguments to options, too. This illustrates some possibilities:

```
_arguments '-option:first arg:->first::optional arg:->second'
```

The doubled colon indicates that the second argument is optional. In other words, at that point on the command line _arguments will either try to complete via the state second, or will try to start another specification entirely.

```
_arguments '-option:first arg:->first:*:other args:->other'
```

Here, all arguments after the first — everything else on the command line — is taken as an argument to the option, to be completed using the state other.

```
_arguments '-option:first arg:->first:*-:other args till -:->other'
```

This is similar, but less drastic: there is a pattern after the '*', here a '-', and when that is encountered, processing of arguments to '-option' stops. A command using this might be called as follows:

```
cmdname -option <first> <other1> <other2> .... - <remainder>
```

where of course completion for `<remainder>` might be handled by other specifications.

There are yet more possible ways of handling options. I've assumed that option names can have multiple letters and hence must occur in separate words. You can specify single-letter options as well, of course, but many commands allow you to combine these into one word. To tell `_arguments` that's OK you should give it the option `-s`; it needs to come before any specifications, to avoid getting mixed up with them. After you specify this, a command argument beginning with a single '`-`' will be treated by `_arguments` as a list of single options, so '`-lt`' is treated the same as '`-l -t`'. However, options beginning with '`--`' are still treated as single options, so a '`--prefix`' on the command line is still handled as a single long option by `_arguments`.

One nice feature which can save a lot of trouble when using certain commands, notably those written by the GNU project and hence installed on most Linux-based systems, which take an option '`--help`' that prints out a list of all options. This is in a human-readable form, but `_arguments` is usually able to extract a list of available options which use the '`--...`' form, and even in many cases whether they take an argument, and if so what type that is. It knows because '`<command> --help`' often prints out a message like '`--file=FILE`' which would tell `_arguments` (1) that '`--file`' is a possible option (2) that it takes an argument because of the '`=`' (3) that that argument should be a file because of the message '`FILE`' at the end.

You specify that the command in question works in this way by using the (fairly memorable) option '`--`' to '`_arguments`'. You can then help it out with completion of option arguments by including a pattern to be matched in the help test after the '`--`'; the format is otherwise similar to a normal specification. For example '`*=FILE*:file:_files`' says that any option with '`=FILE`' in it has the description '`file`' and uses the standard `_files` function for completion, while '`*=DIR*:directory:_files -/`' does the same for directories. These two examples are so common that they are assumed by '`_arguments --`'.

So for example, here is the completion for `gdb`, the GNU debugger, which not surprisingly understands the GNU option format:

```
_arguments -- '*=(CORE|SYM)FILE:core file:_files' \
              '*=EXECFILE:executable:_files -g \*\(\*\)' \
              '*=TTY:terminal device:compadd /dev/tty\*' && return 0
```

If you run '`gdb -help`', you'll see where these come from: '`-core=COREFILE`', '`-exec=EXECFILE`' and '`-tty=TTY`' are all listed as possible option/argument pairs. Doing it this way neatly allows the argument completions to work whatever the names of the options — though of course it's possible for the rest of the pattern to change, too, and the commands, being written by lots of different people, are not necessarily completely consistent in the way their help text is presented.

6.9.8 More completion utility functions

This is now just a ragbag of other functions which might prove useful in your own completion functions, and which haven't been mentioned before, with some examples; once again, consult the manual for more detail. Note that many of these functions can take the most useful arguments to `compadd` and pass them on, even where I haven't explicitly said so.

_call_function

This is a simple front end to calling a function which may not be defined and hanging onto the return status of the function. One good use for this is to call a possibly non-existent function which might have been defined by the user, before doing some default stuff the user might want to skip. That would look like this:

```
local ret  # returned status from called function, if it was called

_call_function ret _hook_function arg1 arg2  &&  return ret

# if we get here, _hook_function wasn't called,
# so do the default stuff.
```

As you can work out, `_call_function` itself returns status zero if the function in the second argument got called, and in that case the first argument is the name of a parameter with the return status from the function itself. The whole point is that this is safe if `_hook_function` doesn't exist.

This function is too low level to know about the tags mechanism; use `_wanted` or similar to handle tags properly.

_contexts

This is another shorthand: the arguments it takes are a set of short contexts, in other words either names of commands or special contexts like '`-math-`'. The completion for each of these contexts is tried in turn; `_contexts` simply handles all the boring looking up of functions and testing the return values. The definition, if you want to look, is reassuringly simple. It only has one use at the moment: `_subscript`, which handles the `-subscript-` context we met early in the chapter, calls '`_contexts -math-`' to try mathematical completion, since ordinary array subscripts can contain mathematical expressions.

This is also too low level to handle tags. In zsh 4.1, it is made obsolete by a cleverer mechanism for handling different contexts which can be used, for example, for handling of arguments to redirections for particular commands, or keys in a particular associative array. I expect I'll describe that when 4.1 is finally released.

_describe

Don't confuse this with `_description` which was explained above and is the basic function for adding a description to a set of completions of a certain type. I mentioned in the description of the `verbose` style that this function was responsible for showing, or not showing, the descriptions for a whole lot of options at once. It allows you to do that with several different sets of completions that may require different options to `compadd`. The general form looks something like this:

```
_describe "description of set 1" descs1 compls1 \
            <compadd-opts-1> -- \
          "description of set 2" ...
```

where you can have any number of sets separated by the '`--`'. The `descs1` and `compls1` are arrays of the same length, giving a list of descriptions and a list of completions, respectively. Alternatively, you need only give one array name and each element of that will contain a completion and a description separated by the now-traditional colon. The '`<compadd-opts-1>`' are a set of any old options recognised by `compadd`, such as `-q`, or `-S=/`, or what have you. I won't give an example for this, since to find something requiring it would almost need me to rewrite the completion system from scratch.

`_combination`

This is the function at the heart of the completions such as `users-hosts` described above, where combinations of elements need to be completed at the same time. It's easiest to describe with an example; let's pick the `users-hosts` example, and I'll assume you remember how that works from the user's point of view, including the format of the `users-hosts` style itself. The completion for the username part is performed as:

```
_combination my-accounts users-hosts users
```

where `my-accounts` is the tag to be used for the completion, then comes the style, and then the part of the style to be extracted.

Now suppose we come back into the completion function again to complete the host later on the command line, so that the username is already there. We can find that by searching the command line; suppose we store what we find in `$userarg`. Then we can complete the hostname as follows:

```
_combination my-accounts users-hosts users=$userarg hosts
```

and the magic part, the fact that we can limit the hostnames to be completed to only those with a user `$userarg`, is handled by `_combination`. This extends to `hosts-ports-users` and any larger combined set in the obvious way: the first field not to contain an '=' is the one being completed. You don't need to supply other fields if they are not known; in other words, the field to be completed doesn't need to be the first one in sequence not known, it can be any, just as long as it matches part of the style given in the second argument, so you could have omitted the '`users=$userarg`' in the last example if you couldn't extract the right username.

There are various bells and whistles: after the field to be completed you can add any options to be passed down to `compadd`; you can give `_combination` itself the option '`-s <sep>`' to specify a character other than colon to separate the parts of the style values; if the style lookup fails, but there is a corresponding function, which would be called '`_users`' or '`_hosts`' in this example, it is called to generate the matches, and gets the options at the end which are otherwise destined for `compadd`.

As you can see, this function is at a high enough level to handle the tags mechanism itself.

`_multi_parts`

This takes two arguments, a separator and a list of matches. The list of matches is normal, except that each element is likely to contain the separator. In the most obvious usage, the separator is '/' and the list of matches is a lot of files with path components. Here's another reasonable usage:

```
local groups expl
groups=($(awk -F: '{ print $1 }' ~/.newsrc))
_wanted groups expl 'newsgroup' _multi_parts "$expl[@]" . groups
```

The generated array contains names of Usenet newsgroups, i.e. names with components separated by a '.', and `_multi_parts` allows you to complete these piece by piece instead of in one go. This is a good deal better for use with menu completion, and the list which appears is smaller too. The `_wanted` part handles the tags mechanism, which `_multi_parts` doesn't.

_sep_parts

This also completes a word piece by piece, but unlike `_multi_parts` the trial completions are also only supplied for each piece. The arguments are alternating arrays and separators; arrays are in the usual form, in other words either the name of an array parameter, or a literal array in parentheses, quoted to protect it from immediate shell expansion. The separators are simply strings. For example

```
local expl
array1=(apple banana cucumber)
_wanted breakfast expl 'breakfast' \
  _sep_parts array1 + '(bread toast croissant)' @ '(bowl plate saucer)';
```

completes strings like '`apple+toast@plate`', piece by piece. This is currently not used by the distributed completion code.

_values

This works a little like `_arguments`, but is designed for completing the values of a single argument in a form like '`key=val,flag,key=other`', in which you can specify the list separator, here ',' by using the option `-s`, e.g. '`-s ,`'. The first argument to `_values` is the overall description of the set of arguments. The other arguments are very much like those to `_arguments` except that, as you would expect from the form given, no pluses or minus signs are involved and each value can only have one argument, which must follow an '='. Virtually everything else is identical, with the exception that the associative array where the arguments are stored for each value is called `$val_args`.

I won't bother giving the instructions for `_arguments` again; instead, here is an example based on the values used by the `-o` option to the `mount` command:

```
local context state line
typeset -A val_args

_values -s , 'file system options' \
  '(rw)ro[mount file system read-only]' \
  '(ro)rw[mount file system read-write]' \
  'uid[set owner of root]:user ID:' \
  'gid[set group of root]:group ID:' \
  'bs[specify block size]:block size:(512 1024 2048 4192)'
```

I've just picked out a few of the umpteen possibilities for illustration; see the function `_mount` if you want more. Remember that the '`(rw)`' before the '`ro`' means that the options are mutually exclusive, and the one in parentheses won't be offered if the other appears on the command line; the strings in square brackets are descriptions of the particular options; and if there is a colon after the name of the value, the value takes an argument whose own description comes next. The second colon is followed by possible completions for that argument, using the usual convention for actions in `_arguments`; as you'll see from the `local` statement, the `$state` mechanism can be used here. Only the '`bs`' argument here is given possible completions; for `uid` and `gid` you'll have to type in the number without completion; `ro` and `rw` don't take arguments.

Hence a typical(?) list to be completed by this would be '`rw,uid=123,bs=2048`'.

Remember also that you can use a '`*`' before the option name to say that it can appear more than once in the value list. The `_values` function handles the context and tags in a similar way to `_arguments`.

`_regex_arguments`

This function is for use when the behaviour of a set of command arguments is so complicated that even `_arguments` can't help. It allows you to describe the arguments as a regular expression (i.e. a pattern). I won't explain it because I haven't yet figured out how it works. If you think you need to use it, look at the manual entry and then at the `_apt` function which is currently its main application.

6.10 Finally

Completion is big and complex: this means that there are probably lots of bugs around, and things that I haven't described simply enough or which may be implemented in too complicated a way. Please send the `zsh-workers` mailing list any reports or constructive criticism on the subject.

Last of all, remember that the new completion system is ideally just supposed to work without you needing to worry exactly how. That's a bold hope, but at least much of the time you should be able to get away with using just the tab key and ordinary characters.

Chapter 7

Modules and other bits and pieces *Not written*

7.1 Control over modules: `zmodload`

7.1.1 Modules defining parameters

7.1.2 Low-level system interaction

7.1.3 ZFTP

7.2 Contributed bits

7.2.1 Prompt themes

7.3 What's new in 4.1

Appendix A

Obtaining zsh and getting more information *Not written*

This appendix gives some useful pointers for finding zsh, more information about zsh, and people who know how to fix the problems (yours or its). It is mostly a set of URLs and email addresses. If you just want to surf your way round, start from the main zsh site at `http://www.zsh.org/`.