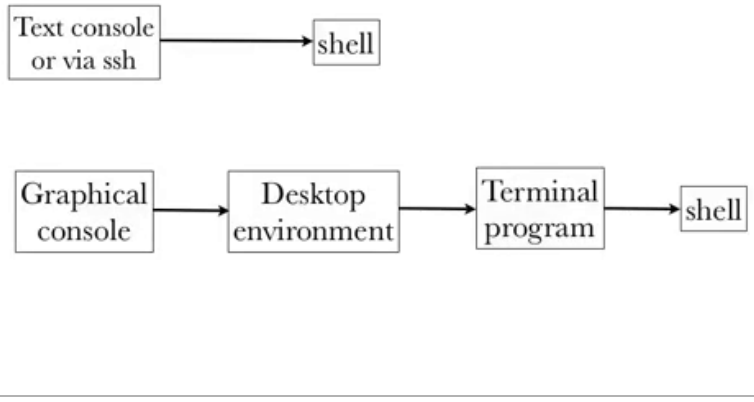
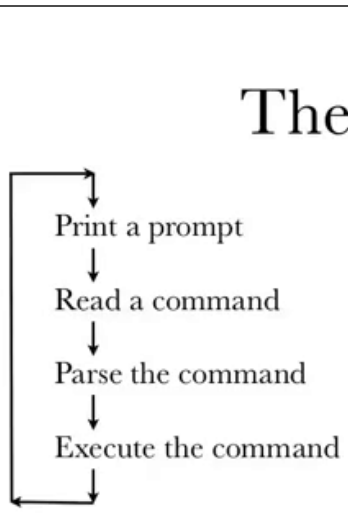
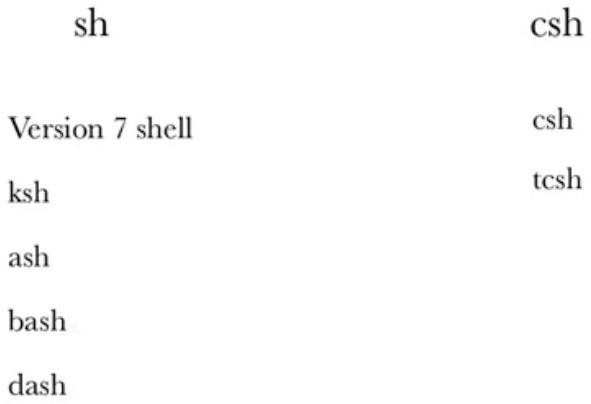
Shell programming 1 of 6

Hover over the image to see the time range in the original video. Click to play that excerpt. Full video at https://www.youtube.com/watch?v=dwf2RBAKh0Q

The Shell	The shell is the command interpreter — the program that you type commands to, that causes them to be executed.
	If you log in from a text console or by ssh or similar, then first it checks your username and password, then it runs a shell. If you log in on a graphical console, it's a bit more complicated — it runs some sort of desktop environment, you have to run a terminal program explicitly, but then the terminal program runs a shell.

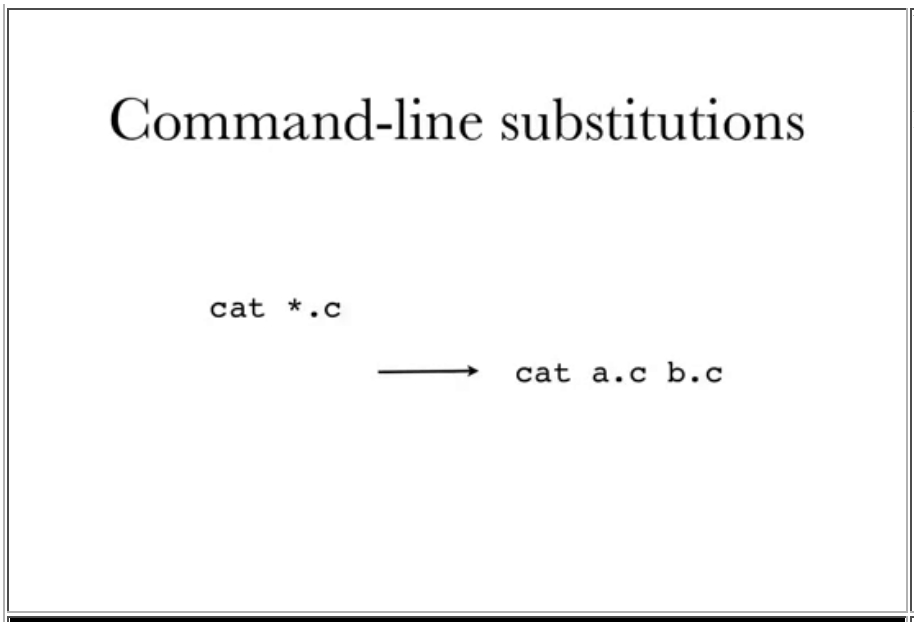


The shell is a big loop. It prints a prompt, which is traditionally a dollar sign and a space; it reads a command; it parses that command; it arranges for that command to be executed; and then it loops, printing another prompt, and so on.



There are two main varieties of shell programming languages, known as sh and csh. Versions of "sh" include the original version 7 unix shell, known as the Bourne shell; ksh, ash, bash, dash, and others. These are all implementations of the basic sh programming language, plus additional features which the author of that particular program thought were a good idea. When we write our shell programs, we will always test them under at least two different implementations of sh, so that we can make sure that we have not inadvertently used any extra features which might not be present when our shell script is run by someone else. Varieties of csh are: csh and tcsh. csh is a greatly inferior programming language. Although some people use csh or tcsh for interactive use, which is fine. But for programming, we will use sh.

You're familiar with typing basic commands to the shell, and pressing return to have them executed. When the shell parses your command-line, it performs various command-line substitutions. For example, filename wildcards are substituted for the matching list of file names by the shell before executing the command.



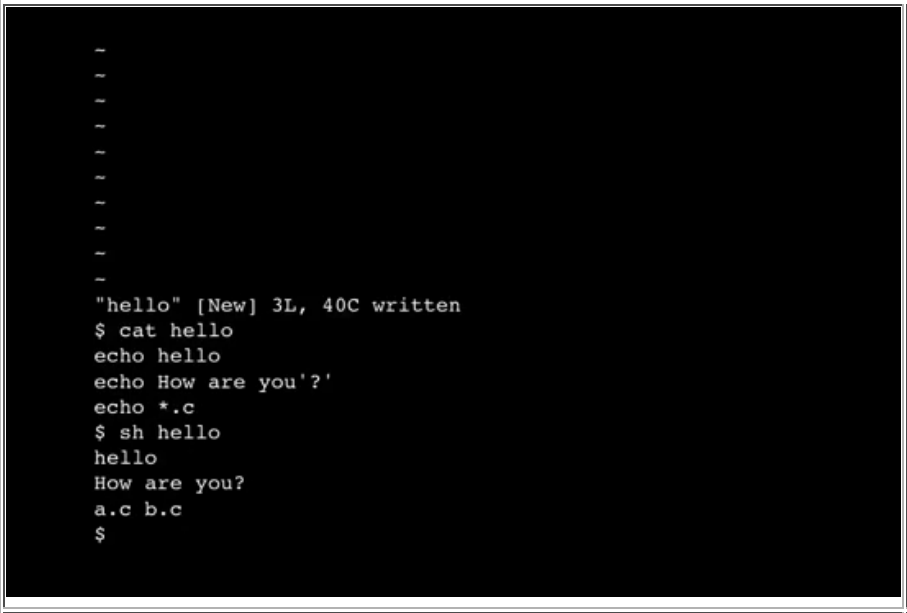
When we type something like "cat *.c", the 'cat' program does not see that "*.c". Instead it sees the expanded argument list based on the list of files in the current directory.

```
$ echo hello
hello
$ echo *.c
a.c b.c
$
```

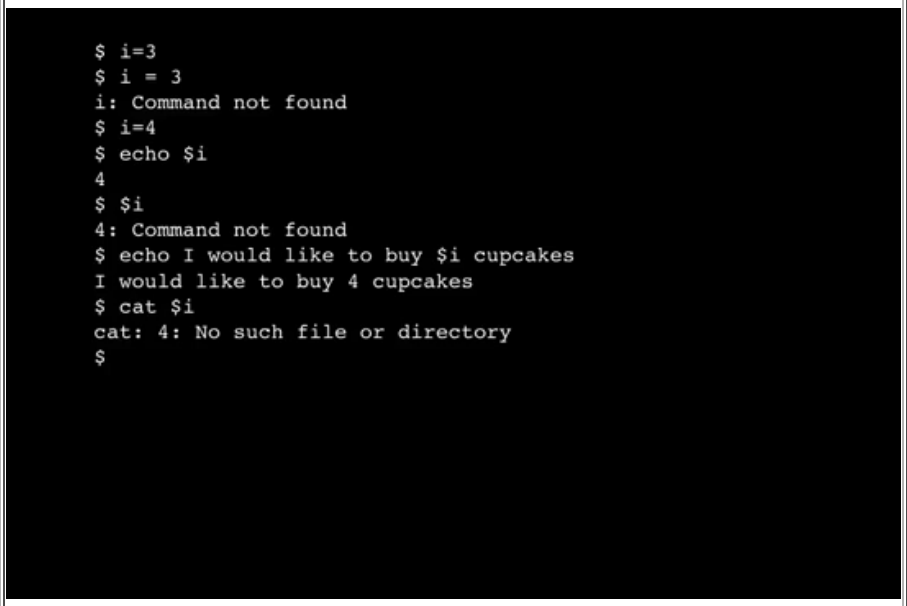
If we want to understand how the shell is doing command-line substitutions, a good way to experiment interactively is with the "echo" command. This is how *echo* works: If you type "echo hello", it will say: "hello". The *echo* command just outputs all of its command-line arguments, separated by spaces. But if we type something like "echo *.c", the shell is going to *interpret* that *.c, expand it to the list of matching file names, and pass *that* to echo *rather* than "*.c". Since echo just outputs its command-line arguments, we get to see what its command-line arguments actually were. So we'll be using the echo command a lot right now to explore how the command-line substitutions work, interactively; although we won't use it as much in our actual programming.

```
echo hello
echo How are you'?'
echo *.c
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"hello" [New] 3L, 40C written
$
```

In fact, *sh* is a complete programming language, although in some ways it's a weird one. This sort of interactive use doesn't look like programming, but we can use sh in a usual software tools way to execute commands from a file, instead of interactively. Let's edit a file using the *vi* editor. I'm going to write "echo hello", "echo How are you'?'" — I have to quote that special character, the question mark, or it will be interpreted as a filename wildcard. "echo *.c" — here I *want* the interpretation; save and quit.



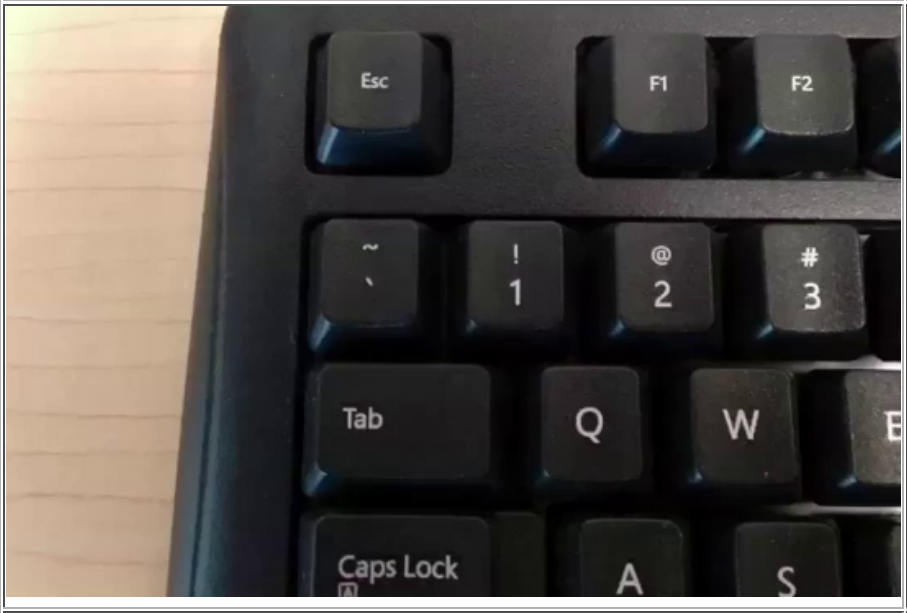
If I run "cat hello", then the *cat* command reads all those lines and outputs them. To *execute* that list of commands, I can instead run the hello file through *sh* instead of *cat* ("sh hello"), and it performs the commands. So the shell works the same when reading commands from a file on disk as when reading commands from the terminal interactively.



Any programming language needs variables. In *sh*, command-line substitutions are involved in using variables. Command-line substitutions are used for many things in sh.
[i=3] This is an assignment statement. And I press return and it's done.
There are no spaces in this assignment statement. "i space equals space 3" (i = 3) means something different: it means to execute a command named i, with first argument equals sign, second argument 3.
i=4
How do we know that this is doing something? How can we explore what the value of i is? If we write a dollar sign and a variable name, then the shell substitutes that, it's a command-line substitution, substitutes that "$i" with the value of the variable i. So "echo $i" is the same as typing "echo 4", and the echo command outputs the 4.
I'm using *echo* here to see how the command-line substitution works. But in practice we would just use $i in a command in some appropriate way. We could even just say "$i" — that tries to execute a command named "4". We can use echo where we would normally use a print statement, and we can use dollar sign i there ("echo I would like to buy $i cupcakes"). Or we can pass it to any other command. If the variable i contained a suitable file name, we could pass it to cat ("cat $i").
[output: cat: 4: No such file or directory]
This is *cat* outputting this error message, because it did run cat. *cat* doesn't see "$i"; it sees the substituted value, as the shell does the command-line substitutions.

```
$ i=i+1
$ echo $i
i+1
$ i=4
$ i=$i+1
$ echo $i
4+1
$ expr 4 + 1
5
$
```

Let's write a statement which increments i. That is to say, i is assigned i+1. First of all, what does "i=i+1" do? Well, if we want to see what it did, one way to see is "echo $i". [output: i+1]
Of course, without the dollar sign this means the string i, not the value of the variable i. But the dollar sign isn't going to help much: if i=4, execute "i=$i+1", now i is "4+1".
No, we need some program which will do the arithmetic. There is a utility program called *expr* which is particularly useful for doing arithmetic in shell programs.
expr, space, 4, space, plus, space, 1 ("expr 4 + 1") outputs 5.
expr is a program which does the arithmetic.

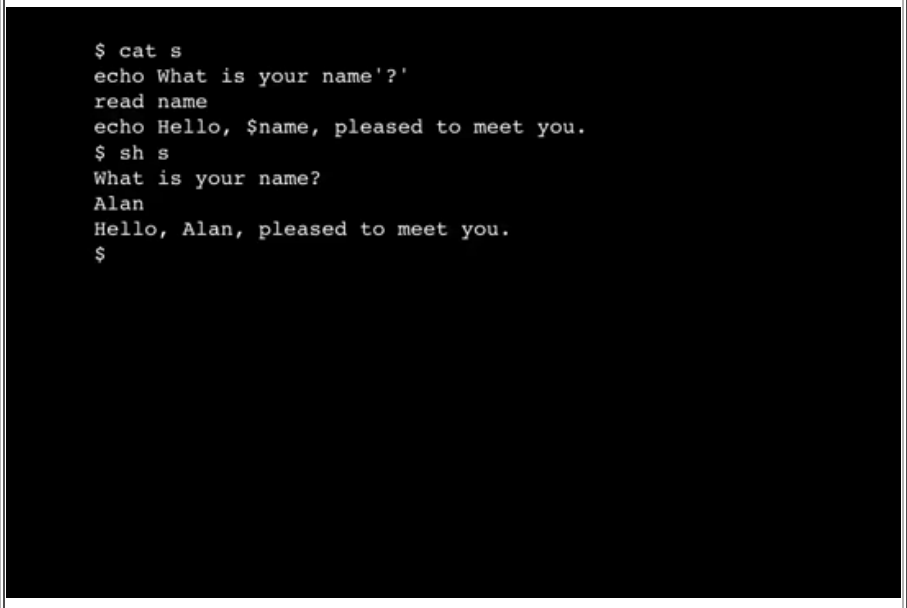


So, how do we apply this to i, to add 1 to i and put it back into i? For this we need another category of command-line substitution. This command-line substitution is invoked with a character called "backquote". The backquote looks like this: `
It's usually on a key in the upper-left of your keyboard.
What is inside backquotes is interpreted as a command, from scratch; that command is executed; its output is *captured*; the output is substituted into this command-line, minus a trailing newline character if any. That definition is a bit overwhelming. Let's play with backquotes a bit.

```
$ i=i+1
$ echo $i
i+1
$ i=4
$ i=$i+1
$ echo $i
4+1
$ expr 4 + 1
5
$ i=`expr 4 + 1`
$ echo $i
5
$ i=`expr $i + 1`
$ echo $i
6
$
```

If we write
`i=`expr 4 + 1``
that is going to run the *expr* command, because that's what backquotes do; it gives it the arguments 4, plus, and 1; *expr*, as we discovered just above, will output the number 5; the backquotes *substitute* that into the command-line, giving us the command "i=5".
How do we know that that's what happened? Well, we could experiment by using the echo command ("echo $i").
If we want to increment i again (i is now 5), we could do
`i=`expr 5 + 1``
But of course we want to write this more generally: all you have to do to do that is to change the 5 to $i.
`i=`expr $i + 1``
Then the shell will execute that *expr* command, which involves a command-line substitution, putting in the 5; "expr 5 + 1" will be executed; the output will be 6, but that will not appear on

	the terminal, it will be captured, because that's what backquotes do; and it will be substituted into the command-line, so we'll have "i=6". Please understand fully how this simple example works, before we move on to more complicated uses of backquotes. "`expr $i + 1`" is a separate command: what backquotes do, it executes that command; it substitutes the output of that command into the command-line. There are no spaces around the equals sign here; as I mentioned in the previous page, that would invoke a command named i. The equals sign has to appear in the first *word* in the command-line. There *are* spaces here ("`expr $i + 1`"). To make it easier to *write* expr, all of the different components are different arguments. The numbers being added together, the operator; each one is a different argument, to make expr easier to write. Please be careful in writing this backquote character on tests! It means something completely different from single quotes. Despite the name, in *sh* backquotes have nothing to do with quoting. Quoting is about suppressing the interpretation of special characters; backquotes are about executing a separate command and substituting its output in on the command-line. Completely different.

	Here's something else we can do with variables. Remember how we can write a simple shell script as a sequence of commands in a file, and then execute it with the sh command.
	I'm going to create a file named "s":
	`echo What is your name'?'`
	`read name`
	`echo Hello, $name, pleased to meet you.`
	read does input.
	Just to see what we're looking at, "cat s" to see the file;
	I'll run it ("sh s");
	It executes the first line of the file, and outputs "what is your name", question mark; question mark is not interpreted as a wildcard character because it's quoted;
	Now it's executing the *read* command. The read command takes a list of variable names. Then I type something; that's going to get assigned to that variable name.

```
echo What is your name'?'
read name
echo Hello, $name, pleased to meet you.
$ sh s
What is your name?
Alan
Hello, Alan, pleased to meet you.
$
$ read x y
foo bar baz
$ echo $x
foo
$ echo $y
bar baz
$ read x y
foo
$ echo $x
foo
$ echo $y

$
```

The read command specifies one or **more** variable names. So if we say "`read x y`"... I'm going to type three different tokens in [typing "`foo bar baz`"]. Each token goes into one corresponding variable. So the "foo" goes into the variable x. If there are more tokens than there are variables listed in the read command, all the rest go into the last variable.
So if you say just one variable, it reads the whole line into that variable.
If there are more variables than tokens, then subsequent variables are assigned to be the empty string.

```
$ PATH=/bin:/usr/bin:/usr/local/bin
$ echo $PATH
/bin:/usr/bin:/usr/local/bin
$
```

Your sh program can use any unix command, including any useful little utilities you write in C — it's quite common for a complex package to include some C code and some shell scripts, and the shell scripts invoke the C programs as needed.
Where does the shell find these commands? It looks through list of directories specified by what's called your "path". There's a special variable named "PATH", in all caps; we can set it to "/bin:/usr/bin:/usr/local/bin"; we can see that it's just a variable; but it's special to the shell, and what it does is, when you type a command which does not contain any slashes in the command name, it looks through the list of directories specified by this variable, separated by colons, and it looks for the command in each of these directories in turn until it finds it.

So that's variables. And a new way to put together commands, with backquotes, which substitute the output of one command as part of another command. But we also need control constructs, like 'if' and 'while' and such, and that's the topic of the second and third shell programming videos.