8 — Using the Shell

8.1. Programs are tools

In Plan 9, programs are tools that can be combined to perform very complex tasks. In most other systems, the same applies, although it tends to be a little more complex. The idea is inherited from UNIX, each program is meant to perform a single task, and perform it well.

But that does not prevent you to combine existing programs to do a wide variety of things. In general, when there is a new job to be done, these are your options, listed from the easiest one to the hardest one:

1. Find a program that does the job. It is utterly important to look at the manual before doing anything. In many cases, there will be a program that does what we want to do. This also applies when programming in C, there are many functions in the library that may greatly simplify your programs.
2. Combine some programs to achieve the desired effect. This is where the shell gets relevance. The shell is the programming language you use to combine the programs you have in a simple way. Knowing how to use it may relieve you from your last resort.
3. The last resort is to write your own program for doing the task you are considering. Although the libraries may prove invaluable as helpers, this requires much more time, specially for debugging and testing.

To be able to use shell effectively, it helps to follow conventions that may be useful for automating certain tasks by using simple shell programs. For example, writting each C function using the style

```
void
func(...args...)
{
}
```

permits using this command line to find where function `foo` is defined:

```
; grep -n  '^foo\(' *.c
```

By convention, we declared functions by writing their names at the beggining of a new line, immediatelly followed by the argument list. As a result, we can ask `grep` to search for lines that have a certain name at the beggining of line, followed by an open parenthesis. And that helps to quickly locate where a function is defined.

The shell is very good for processing text files, and even more if the data has certain regularities that you may exploit. The shell provides a full programming language where commands are to be used as elementary statements, and data is handled in most cases as plain text.

In this chapter we will see how to use Rc as a programming language, but no one is going to help you if you don't help yourself in the first place. Machines love regular structures, so it is better to try to do the same thing in the same way everywhere. If it can be done in a way that can simplify your job, much better.

Plan 9 is a nice example of this is practice. Because all the resources are accessed using the same interface (a file interface), all the programs that know how to do particular things to files can be applied for all the resources in the system. If many different interfaces were used instead, you would need many different tools for doing the same operation to the many different resources you find in the computer.

This explains the popularity of XML and other similar data representations, which are attempts to provide a common interface for operating on many different resources. But the idea is just the same.

8.2. Lists

The shell includes lists as its primary data structure, as its only data structure, indeed. This data type is there to make it easier for you to write shell programs. Because shell variables are just environment variables, lists are stored as strings, the only value a environment variable may have. This is the famous abc list:

```
; x=(a b c)
; echo $x
a b c
```

It is just syntax. It would be the same if we had typed any of the following:

```
; x=(a (b c))
; echo $x
a b c
; x=(((a) (b)) (c))
; echo $x
a b c
```

It does not matter how you nest the same values using multiple parenthesis. All of them will be the same, namely, just `(a b c)`. What is the actual value of the environment variable for `x`? We can see it.

```
; xd -c /env/x
0000000   a 00  b 00  c 00
0000006
```

Just the three strings, `a`, `b`, and `c`. Rc follows the C convention for terminating a string, and separates all the values in the list with a null byte. This happens even for environment variables that are a list of a single word.

```
; x=3
; xd -c /env/x
0000000   3 00
0000002
```

The implementation for the library function `getenv` replaces the null bytes with spaces, and that is why a `getenv` for an Rc list would return the words in the list separated by white space. This is not harmful for C, as a 0 would be because 0 is used to terminate a string in C. And it is what you expect after using the variable in the shell.

The variable holding the arguments for the shell interpreting a shell script is also a list. The only difference is that the shell initializes the environment variable for `$*` automatically, with the list for the arguments supplied to it, most likely, by giving the arguments to a shell script.

Given a variable, we can know its length. For any variable, the shell defines another one to report its length. For example,

```
; x=hola
; echo $#x
1
; x=(a b c)
; echo $#x
3
```

The first variable was a list with just one word in it. As a result, this is the way to print the number of arguments given to a shell script,

```
echo $#*
```

because that is the length of `$*`, which is a list with the arguments (stored as an environment variable).

To access the *n*-th element of a list, you can use `$var(n)`. However, to access the *n*-th argument in a shell script you are expected to use `$n`. An example for our popular abc list follows:

```
; echo $x(2)
b
; echo $x(1)
a
```

Lists permit doing funny things. For example, there is a concatenation operator that is best shown by example.

```
; x=(a b c)
; y=(1 2 3)
echo $x^$y
a1 b2 c3
```

The `^` operator, used in this way, is useful to build expressions by building separate parts (e.g, prefixes and suffixes), and then conbining them. For example, we could write a script to adjust permissions that might set a variable `ops` to decide if we should add or remove a permission, and then a variable `perms` to list the involved permissions. Of course in this case it would be easier to write the result by hand. But, if we want to generate each part separately, now we can:

```
; ops=(+ - +)
; perms=(r w x)
; echo $ops^$perms afile
+r -w +x afile
```

Note that concatenating two variables of length 1 (i.e., with a single word each) is a particular case of what we have just seen. Because this is very common, the shell allows you to omit the `^`, which is how you would do the same thing when using a UNIX shell. In the example below, concatenating both variables is *exactly* the same than it would have been writing `a1` instead.

```
; x=a
; y=1
; echo $x^$y
a1
; echo $x$y
a1
;
```

A powerful use for this operator is concatenating a list with another one that has a single element. It saves a lot of typing. Several examples follow. We use `echo` in all of them to let you see the outcome.

```
; files=(stack run cp)
; echo $files^.c
stack.c run.c cp.c
; echo $files^.h
stack.h run.h cp.h
; rm $files^.8
; echo (8 5)^.out
8.out 5.out
; rm (8 5)^.out
```

Another example. These two lines are equivalent:

```
; cp (/source/dir /dest/dir)^/a/very/long/path
; cp /source/dir/a/very/long/path /dest/dir/a/very/long/path
```

And of course, we can use variables here:

```
; src=/source/dir
; dst=/dest/dir
; cp ($src $dst)^/a/very/long/path
```

Concatenation of lists that do not have the same number of elements and do not distribute, because none of them has a single element, is illegal in Rc. Contatenation of an empty list is also forbidden, as a particular case of this rule.

```
; ops=(+ - +)
; perms=(w x)
; echo $ops^$perms
rc: mismatched list lengths in concatenation
; x=()
; echo (a b c)^$x
rc: null list in concatenation
```

In some cases it is useful to use the value of a variable as a single string, even if the variable contains a list with several strings. This can be done by using a "" before the variable name. Note that this may be used to concatenate a variable that might be an empty list, because we translate the variable contents to a single word, which happens to be empty.

```
; x=(a b c)
; echo $x^1
a1 b1 c1
; echo $"x^1
a b c1
; x=()
; echo (a b c)^$"x
a b c
;
```

There are two slightly different values that can be used to represent a null variable. One is the empty string, and the other one is the empty list. Here they are, in that order.

```
; x=''
; y=()
; echo $x

; echo $y

; xd -c /env/x
0000000  00
0000001
; xd -c /env/y
0000000
0000000
; echo $#x $#y
1 0
```

Both values yield a null string when used, yet they are different. An empty string is a list with just the empty string. When expanded by `getenv` in a C program, or by using `$` in the shell, the result is the empty string. However, its length is 1 because the list has one (empty) string. For an empty list, the length is zero. In general, it is common to use the empty list as the nil value for environment variables.

8.3. Simple things

We are now prepared to start doing useful things. To make a start, we want to write a couple of shell scripts to convert from decimal to hexadecimal and vice-versa. We should start most scripts with

```
rfork e
```

to avoid modifying the set of environment variables in the calling shell.

The first thing needed is a program to perform arithmetic calculations. The shell knows *nothing* about numbers, not to talk about arithmetic. The shell knows how to combine commands together to do useful work. Therefore, we need a program to do arithmetic if we want to do arithmetic with the shell. We may type numbers, but for shell, they would be just strings. Lists of strings indeed. Let's search for that program.

```
; lookman arithmetic expression
man 1 2c # 2c(1)
man 1 awk # awk(1)
man 1 bc # bc(1)
man 1 hoc # hoc(1)
man 1 test # test(1)
man 8 prep # prep(8)
```

There are several programs shown in this list that we might use to do arithmetic. In general, `hoc` is a very powerful interactive floating point calculation language. It is very useful to compute arbitrary expressions, either by supplying them through its standard input or by using its `-e` option, which accepts as an argument an expression to evaluate.

```
; hoc -e '2 + 2'
4
; echo 2 + 2 | hoc
4
```

Hoc can do very complex arithmetic. It is a full language, using a syntax similar to that of C. It reads expressions, evaluates them, and prints the results. The program includes predefined variables for famous constants, with names `E`, `PI`, `PHI`, etc., and you can define your own, using the assignment. For example,

```
; hoc
r=3.2
PI * r^2
32.16990877276
control-d
;
```

defines a value for the radius of a circle, and computes the value for its area.

But to do the task we have at hand, it might be more appropriate another calculation program, called `bc`. This is program is also a language for doing arithmetic. The syntax is also similar to C, and it even allows to define functions (like Hoc). Like before, this tool accepts expressions as the input. It evaluates them and prints the results. The nice thing about this program is that it has a simple way of changing the numeric base used for input and output. Changing the value for the variable `obase` changes the base used for output of numeric values. Changing the value for the variable `ibase` does the same for the input. It seems to be just the tool. Here is a session converting some decimal numbers to hexadecimal.

```
; bc
obase=16
10
a
20
14
16
10
```

To print a decimal value in hexadecimal, we can write `obase=16` and the value as input for `bc`. That would print the desired output. There are several ways of doing this. In any case, we must send several statements as input for `bc`. One of them changes the output base, the other prints the desired value. What we can do is to separate both `bc` statements with a "`;`", and use `echo` to send them to the standard input of `bc`.

```
; echo 'obase=16 ;  512' | bc
200
```

We had to quote the whole command line for `bc` because there are at least two characters with special meaning for Rc, and we want the string to be echoed verbatim. This can be packated in a shell script as follows, concatenating `$1` to the rest of the command for `bc`.

|d2h|

```
#!/bin/rc
echo 'obase=16;  '$1 | bc
```

Although we might have inserted a `^` before `$1`, Rc is kind enough to insert one for free for us. You will get used to this pretty quickly. We can now use the resulting script, after giving it execute permission.

```
; chmod +x d2h
; d2h 32
20
```

We might like to write each input line for `bc` using a separate line in the script, to improve readability. The compound `bc` statement that we have used may become hard to read if we need to add more things to it. It would be nice to be able to use a different `echo` for each different command sent to `bc`, and we can do so. However, because the output for *both* echoes must be sent to the standard input of `bc`, we must group them. This is done in Rc by placing both commands inside brackets. We must still quote the first command for `bc`, because the equal sign is special for Rc. The resulting script can be used like the one above, but this one is easier to read.

```
#!/bin/rc
{       echo 'obase=16'
        echo $1
} | bc
```

Here, the shell executes the two `echo`es but handles the two of them as it they were just one command, regarding the redirection of standard output. This grouping construct permits using several commands wherever you may type a single command. For example,

```
; { sleep 3600 ; echo time to leave! } &
;
```

executes *both* `sleep` and `echo` in the background. Each command will be executed one after another, as expected. The result is that in one hour we will see a message in the console reminding that we should be leaving.

How do we implemente a script, called `h2d`, to do the opposite conversion? That is, to convert from hexadecimal to decimal. We might do a similar thing.

```
#!/bin/rc
{       echo 'ibase=16'
        echo $1
} | bc
```

But this has problems!

```
; h2d abc
syntax error on line 1, teletype
syntax error on line 1, teletype
0
```

The problem is that `bc` expects hexadecimal digits from `A` to `F` to be uppercase letters. Before sending the input to `bc`, we would better convert our numbers to uppercase, just in case. There is a program that may help. The program `tr` translates characters. It reads its input files (or standard input), performs its simple translations, and writes the result to the output. The program is very useful for doing simple character transformations on the input, like replacing certain characters with other ones, or removing them. Some examples follow.

```
; echo x10+y20+z30 | tr x y
y10+y20+z30
; echo x10+y20+z30 | tr xy z
z10+z20+z30
; echo x10+y20+z30 | tr a-z A-Z
X10+Y20+Z30
; echo x10+y20+z30 | tr -d a-z
10+20+30
```

The first argument states which characters are to be translated, the second argument specifies to which ones they must be translated. As you can see, you can ask `tr` to translate several different characters into a single one. When many characters are the source or the target for the translation, and they are contiguous, a range may be specified by separating the initial and final character with a dash. Under option `-d`, `tr` removes the characters from the input read, before copying the data to the output. So, how would translate a dash to other character? Simple.

```
; echo a-b-c | tr - X
aXbXc
```

This may be a problem we need to translate some other character, because `tr` would get confussed thinking that the character is an option.

```
; echo a-b-c | tr -a XA
tr: bad option
```

But this can be fixed reversing the order for characters in the argument.

```
; echo a-b-c | tr a- AX
AXbXc
```

Now we can get back to our `h2d` tool, and modify it to supply just uppercase hexadecimal digits to `bc`.

|h2d|

```
#!/bin/rc
{       echo 'ibase=16'
        echo print $1 | tr a-f A-F
} | bc
```

The new `h2d` version works as we could expect, even when we use lower-case hexadecimal digits.

```
; h2d abc
2748
```

Does it pay to write `h2d` and `d2h`? Isn't it a lot more convenient for you to use your desktop calculator? For converting just one or two numbers, it might be. For converting a docen or more, it is for sure it pays to write the script. The nice thing about having one program to do the work is that we can now use the shell to automate things, and let the machine work for us.

8.4. Real programs

Our programs `h2d` and `d2h` are useful, for a casual use. To use them as building blocks for doing more complex things, more work is needed. Imagine you need to declare an array in C, and initialize it, to use the array for translating small integers to their hexadecimal representation.

```
char* d2h[] = {
        "0x00",
        "0x11",
        …
        "0xff"
};
```

To obtain a printable string for a integer `i` in the range 0-255 you can use just `d2h[i]`. Would you write that declaration by hand? No. The machine can do the work. What we need is a command that writes the first 256 values in hexadecimal, and adjust the output text a little bit before copying it to your editor.

We could change `d2h` to accept more than one argument and do its work for *all* the numbers given as argument. Calling `d2h` with all the numbers from 0 to 255 would get us close to obtaining an initializer for the array. But first things first. We need to iterate through all the command line arguments in our script. Rc includes a `for` construct that can be used for that. It takes a variable name and a list, and executes the command in the body once for each word in the list. On each pass, the variable takes the value of the corresponding word. This is an example, using `x` as the variable and `(a b c)` as the list.

```
; for (x in a b c)
;; echo $x
a
b
c
```

Note how the prompt changed after typing the `for` line, Rc wanted more input: The command for the body. To use more than one command, we may use the brackets as before, to group them. First attempt:

```
; for (num in 10 20 30) {
;; echo 'obase=16'
;; echo $num
;; }
obase=16
10
obase=16
20
obase=16
30
;
```

It is useful to try the commands before using them, to see what really happens. The `for` loop gave three passes, as expected. Each time, `$num` kept the value for the corresponding string in the list: `10`, `20`, and `30`. Remember, these are strings! The shell does not know they mean numbers

to you. Setting `obase` in each pass seems to be a waste. We will do it just once, before iterating through the numbers. The numbers are taken from the arguments given to the script, which are kept at `$*`.

d2h

```
#!/bin/rc
rfork e
{
        echo 'obase=16'
        for (num in $*)
                echo $num
} | bc
```

Now we have a better program. It can be used as follows.

```
; d2h 10 20 40
a
14
28
```

We still have the problem of supplying the whole argument list, a total of 256 numbers. It happens that another program, `seq`, knows how to write numbers in sequence. It can do much more. It knows how to print numbers obtained by iterating between two numbers, using a certain step.

```
; seq 5  from 1 to 5
1
2
3
4
5

; seq 1 2 10      from 1 to 10 step 2
1
3
5
7
9
;
```

What we need is to be able to use the output of `seq` as an argument list for `d2h`. We can do so! When Rc finds a command inside `'{...}`, it executes the command, and *substitutes* the whole `'{...}` text with the output printed by the command. We did something alike in a C program when reading the output for a command using a pipe. This time, Rc does it for us, and relieves us from typing something that can be generated using a program. This is an example.

```
; seq 1 5
1
2
3
4
5
; echo '{seq 1 5}
1 2 3 4 5
;
```

As you can see, the second command was equivalent to this one:

```
; echo 1 2 3 4 5
```

The shell executed `seq 1 5`, and then did read the text printed by this command in standard output. Once all the command output was read, Rc replaced the whole `‘{...}` construct with the text just read. The resulting line was the one executed, intead of the one that we originaly typed. Because a newline character terminates a command, the shell replaced each `\n` in the command output with a space. That is why executing `seq` directly yields 5 lines of output, but using it with `‘{...}` produces just one line of output. We can do now what we wanted.

```
; d2h ‘{seq 1 255}
1
2
...and many other numbers up to...
fd
fe
ff
```

That was nice. However, most programs that accept arguments, work with their standard input when no argument is given. If we do the same to `d2h`, we increase the opportunities to reuse it for other tasks. The idea is simple, we must check if we have arguments. If there are some, we proceed as before. Otherwise, we can read the arguments using `cat`, and then proceed as before. We need a way to decide what to do, and we need to be able to compare things. Rc provides both things.

The construction `if` takes a command as an argument (whitin parenthesis). If the command’s exit status is all right (i.e., the empty string), the body is executed. Otherwise, the body is not executed. This is the classical *if-them*, but using a command as the condition (which makes sense for a shell), and one command (or a group of them) as a body.

```
; if (ls -d /tmp) echo /tmp is there!
/tmp
/tmp is there!
;
; if (ls -d /blah) echo blah is there
ls: /blah: ’/blah’ file does not exist
```

In the first case, Rc executed `ls -d /tmp`. This command printed the first output line, and, because its exit status was the empty string, it was taken as *true* regarding the condition for the `if`. Therefore, `echo` was executed and it printed the second line. In the second case, `ls -d /blah` failed, and `ls` complained to its standard error. The body command for the `if` was not executed.

It can be a burden to see the output for commands that we use as conditions for `if`s, and it may be wise to send the command output to `/dev/null`, including its standard error.

```
; if (ls -d /tmp >/dev/null >[2=1]) echo is there
is there
; if (ls -d /blah >/dev/null >[2=1]) echo is there
;
```

Once we know how to decide, how can we compare strings? The ~ operator in Rc compares one string to other ones[1], and yields an exit status meaning true, or success, when the comparation succeeds, and one meaning false otherwise.

[1] We will see how ~ is comparing a string to expressions, not just to strings.

```
; ~ 1 1
; echo $status

; ~ 1 2
; echo $status
no match
; if (~ 1 1) echo this works
this works
```

So, the plan is as follows. If `$#*` (the number of arguments for our script) is zero, we must do something else. Otherwise, we must execute our previous commands in the script. Before implementing it, we are going to try just to do different things depending on the number of arguments. But we need an else! This is done by using the construct `if not` after an `if`. If the command representing the condition for an `if` fails, the following `if not` executes its body.

|args|

```
#!/bin/rc
if (~ $#* 0)
        echo no arguments
if not
        echo got some arguments: $*
```

And we can try it.

```
; args
no arguments
; args 1 2
got some arguments: 1 2
```

Now we can combine all the pieces.

|d2h|

```
#!/bin/rc
rfork e
if (~ $#* 0)
        args=‘{cat}
if not
        args=$*
{
        echo ’obase=16’
        for (num in $args)
                echo $num
} | bc
```

We try our new script below. When using its standard input to read the numbers, it uses the `‘{...}` construct to execute `cat`, which reads all the input, and to place the text read in the environment variable `args`. This means that it will not print a single line of output until we have typed all the numbers and used *control-d* to simulate an end of file.

```
; d2h3
20
30
control-d
14
1e
;
; d2h3 3 4
3
4
;
```

Our new command is ready for use, and it can be combined with other commands, like in `seq 10|d2h`. It would work as expected.

An early exercise in this book asked to use `ip/ping` to probe for all addresses for machines in a local network. Addresses were of the form `212.128.3.X` with `X` going from 1 to 254. You now know how to it fast!

```
; nums='{seq 1 254}
; for (n in $nums) ip/ping 212.128.3.$n
```

Before this example, you might have been saying: Why should I bother to write several shell command lines to do what I can do with a single loop in a C program? Now you may reconsider the question. The answer is that in Rc it is very easy to combine commands. Doing it in C, that is a different business.

By the way. Use variables! They might save a lot of typing, not to talk about making commands more simple to read. For instance, the next commands may be better than what we just did. If we have to use `212.128.3` again, which is likely if we are playing with that network, we might just say `$net`.

```
; nums='{seq 1 254}
; net=212.128.3.
; for (n in $nums) ip/ping $net^$n
```

8.5. Conditions

Let's go back to commands used for expressing conditions in our shell programs. The shell operator ~ uses expressions. They are the same expressions used for globbing. The operator receives at least two arguments, maybe more. Only the first one is taken as a string. The remaining ones are considered as expressions to be matched against the string. For example, this iterates over a set of files and prints a string suggesting what the file might be, according to the file name.

file

```
#!/bin/rc
rfork e
for (file in $*) {
        if (~ $file *.c *.h)
                echo $file: C source code
        if (~ $file *.gif)
                echo $file: GIF image
        if (~ $file *.jpg)
                echo $file: JPEG image
}
```

And here is one usage example.

```
; file x.c a.h b.gif z
x.c: C source code
a.h: C source code
b.gif: GIF image
```

Note that before executing the ~ command, the shell expanded the variables, and `$file` was replaced with the correponding argument on each pass of the loop. Also, because the shell knows that ~ takes expressions, it is not necessary to quote them. Rc does it for you.

There is a `switch` construct in Rc that permits doing multiway branches, like the construct of the same name in C. The one of Rc takes one string as the argument, and executes the branch with a regular expression that matches the string. Each branch is labeled with the word `case` followed by the expressions for the branch. This is an example that improves the previous script.

```
#!/bin/rc
rfork e
for (file in $*) {
        switch($file){
        case *.c *.h
                echo $file: C source code
        case *.gif
                echo $file: GIF image
        case *.jpg
                echo $file: JPEG image
        case *
                echo $file: who knows
        }
}
```

As you can see, in a single `case` you may use more than one expression, like you can with ~. As a matter of fact, this script is doing poorly what is better done with a standard command that has the same name, `file`. This command prints a string after inspecting each file whose name is given as an argument. It reads each file to search for words or patterns and makes an educated guess.

```
; file ch7.ms ch8.ps src/hi.c
ch7.ms:   Ascii text
ch8.ps:   postscript
src/hi.c: c program
```

There is another command that was built just to test for things, to be used as a condition for `if` expressions in the shell. This program is `test`. For example, the option `-e` can be used to check that a file does exist, and the option `-d` checks that a file is a directory.

```
; test -e /LICENSE
; echo $status

; test -e /blah
; echo $status
test 52313: false
; if (test -d /tmp) echo yes
yes
; if (test -d /LICENSE) echo yes
;
```

8.6. Editing text

Before, we managed to generate a list of numbers for an array initializer that we did *not* want to write by ourselves. But the output we obtained was not yet ready for a cut-and-paste into our editor. We need to convert something like

```
1
2
...
```

into something like

```
"0x1",
"0x2",
...
```

that can be used for our purposes. There are many programs that operate on text and know how to do complex things to it. In this section we are going to explore them.

To achieve our purpose, we might convert each number into hexadecimal, and store the resulting string in a variable. Later, it is just a matter of using `echo` to print what we want, like follows.

```
; num=32
; hexnum=‘{{ echo ’obase=16’ ; echo $num } | bc}
; echo "0x^$hexnum^",
"0x20",
```

We used the `‘{...}` construct execute `hexnum=...`, with the appropriate string on the right hand side of the equal sign. This string was printed by the command

```
{ echo ’obase=16’ ; echo $num } | bc
```

that we now know that prints `20`. It is the same command we used in the `d2h` script.

For you, the “`"`” character may be special. For the shell, it is just another character. Therefore, the shell concatenated the “`"0x`” with the string from `$hexnum` and the string “`",`”. That was the argument given to `echo`. So, you probably know already how to write a few shell command lines to generate the text for your array initializer.

```
; for (num in ‘{seq 0 255}) {
;;      number=‘{{ echo ’obase=16’ ; echo $num } | bc}
;;      echo "0x^$number^",
;; }
"0x0",
"0x1",
"0x2",
```
...and many others follow.

Is the problem solved? Maybe. This is a very innefficient way of doing things. For each number, we are executing a couple of processes to run `echo` and then another process to run `bc`. It takes time for processes to start. You know what `fork` and `exec` do. That must take time. Processes are cheap, but not free. Wouldn’t it be better to use a single `bc` to do all the computation, and then adjust the output? For example, this command, using our last version for `d2h`, produces the same output. The final `sed` command inserts some text at the beggining and at the end of each line, to get the desired output.

```
; seq 1 255 | d2h | sed -e ’s/^/"0x/’ -e ’s/$/",/’
"0x0",
"0x1",
"0x2",
```
...and many others follow.

To see the difference betwen this command line, and the direct `for` loop used above, we can use `time` to measure the time it takes to each one to complete. We placed the command above using a `for` into a `/tmp/for` script, and the last command used, using `sed`, at a script in `/tmp/sed`. This is what happen.

```
; time /tmp/sed >/dev/null
0.34u 1.63s 5.22r        /tmp/sed
; time /tmp/for >/dev/null
3.64u 24.38s 74.30r      /tmp/for
```

The `time` command uses the `wait` system call to obtain the time for its child (the command we want to measure the time for). It reports the time spent by the command while executing user code, the time it spent while inside the kernel, executing system calls and the like, and the real (elapsed) time until it completed. Our loop, starting several processes for each number being processed, takes 74.3 seconds to generate the output we want! That is admiteddly a lot shorter than doing it by hand. However, the time needed to do the same using `sed` as a final processing step in the pipeline is just 5.22 seconds. Besides, we had to type less. Do you think it pays?

The program `sed` is a *stream editor*. It can be used to edit data as it flows through a pipeline. Sed reads text from the input, applies the commands you give to edit that text, and writes the result to the output. In most cases, this command is used to perform simple tasks, like inserting, deleting, or replacing text. But it can be used for more. As with most other programs, you may specify the input for `sed` by giving some file names as arguments, or you may let it work with the standard input otherwise.

In general, editing commands are given as arguments to the `-e` option, but if there is just one command, you may omit the `-e`. For example, this prints the first 3 lines for a file.

```
; sed 3q /LICENSE
The Plan 9 software is provided under the terms of the
Lucent Public License, Version 1.02, reproduced below,
with the following notable exceptions:
;
```

All sed commands have either none, one, or two *addresses* and then the command itself. In the last example there was one addres, `3`, and one command, `q`. The editor reads text, usually line by line. For each text read, `sed` applies all the editing commands given, and copies the result to standard output. If addresses are given for a command, the editor applies the command to the text selected by those addresses.

A number is an address that corresponds to a line number. The command `q`, quits. What happen in the example is that the editor read lines, and printed them to the output, until the address `3` was matched. That was at line number 3. The command *quit* was applied, and the rest of the file was not printed. Therefore, the previous command can be used to print the first few lines for a file.

If we want to do the opposite, we may just *delete* some lines, from the one with address 1, to the one with address 3. As you can see below, both addresses are separated with a comma, and the command to apply follows. Therefore, `sed` searched for the text matching the address pair `1,3` (i.e., lines 1 to 3), printing each line as it was searching. Then it copied the text selected to memory, and applied the `d` command. These lines were deleted. Afterwards, `sed` continued copying line by line to its memory, doing nothing to each one, and copying the result to standard output.

```
; sed 1,3d /LICENSE

1. No right is granted to create derivative works of or
   to redistribute (other than with the Plan 9 Operating System)
...more useful stuff for your lawyer...
```

Supplying just one command, with no address, applies the command to all lines.

```
; sed d /LICENSE
;
```

Was the `/LICENSE` deleted? Of course not. This editor is a *stream* editor. It reads, applies commands to the text while in the editor's memory, and outputs the resulting text.

How can we print the lines 3 to 5 from our input file? One strategy is to use the `sed` command to print the text selected, `p`, selecting lines 3 to 5. And also, we must ask `sed` not to print lines by default after processing them, by giving the `-n` flag.

```
; sed -n 3,5p /LICENSE
with the following notable exceptions:

1. No right is granted to create derivative works of or
```

The special address `$` matches the end of the file. Therefore, this deletes from line 3 to the end of the file.

```
; sed '3,$d' /LICENSE
The Plan 9 software is provided under the terms of the
Lucent Public License, Version 1.02, reproduced below,
```

What follows deletes lines between the one matching `/granted/`, i.e., the first one that contains that word, and the end of the file. This is like using `1,3d`. There are two addresses and a `d` command. It is just that the two addresses are more complicated this time.

```
; sed '/granted/,$d' /LICENSE
The Plan 9 software is provided under the terms of the
Lucent Public License, Version 1.02, reproduced below,
with the following notable exceptions:

;
```

Another interesting command for `sed` is `r`. This one reads the contents of a file, and writes them to the standard output before proceeding with the rest of the input. For example, given these files,

```
; cat salutation
Today I feel
FEEL
So be warned
; cat how
Really in bad mood
;
```

we can use `sed` to adjust the text in `salutation` so that the line with `FEEL` is replaced with the contents of the file `how`. What we have to do is to give `sed` an address that matches a line with the text `FEEL` in it. Then, we must use the `d` command to delete this line. And later we will have to insert in place the contents of the other file.

```
; sed /FEEL/d <salutation
Today I feel
So be warned
```

The address `/FEEL/` matches the string `FEEL`, and therefore selects that line. For each match, the command `d` removes its line. If there were more than one line matching the address, all of such lines would have been deleted. In general, `sed` goes line by line, doing what you want.

```
; cat salutation  salutation | sed /FEEL/d
Today I feel
So be warned
Today I feel
So be warned
```

We also wanted to insert the text in `how` in place, besides deleting the line with `FEEL`. Therefore, we want to execute *two* commands when the address `/FEEL/` matches in a line in the input. This can be done by using braces, but `sed` is picky regarding the format of its program, and we prefer to use several lines for the `sed` program. Fortunately, Rc knows how to quote it all.

```
; sed  -e '/FEEL/{
;; r how
;; d
;; }'<salutation
Today I feel
Really in bad mood
So be warned
```

In general, it is a good idea to quote complex expressions that are meant not for shell, but for the command being executed. Otherwise, we might use a character with special meaning for Rc, and there could be surprises.

This type of editing can be used to prepare templates for certain files, for example, for your web page, and then automatically adjust this template to generate something else. You can see the page at `http://lsub.org/who/nemo`, which is generated using a similar technique to state whether Nemo is at his office or not.

The most useful `sed` command is yet to be seen. It replaces some text with another. Many people who do not know how to use `sed`, *know* at least how to use `sed` just for doing this. The command is `s` (for *substitute*), and is followed by two strings. Both the command and the strings are delimited using any character you please, usually a `/`. For example, `s/bad/good/` replaces the string `bad` with `good`.

```
; echo Really in bad mood | sed 's/bad/good/'
Really in good mood
```

The quoting was unnecessary, but it does not hurt and it is good to get used to quote arguments that may get special characters inside. There are two things to see here. The command, `s`, applies to *all* lines of input, because no address was given. Also, as it is, it replaces only the first apearance of `bad` in the line. Most times you will add a final `g`, which is a flag that makes `s` substitute all occurrences (globally) and not just the first one.

This lists all files terminating in `.h`, and replaces that termination with `.c`, to generate a list of files that may contain the implementation for the things declared in the header files.

```
; ls *.h
cook.h
gui.h
; ls *.h | sed 's/.h/.c/g'
cook.c
gui.c
```

You can now do more things, like renaming all the files terminated in `.cc` to files terminated in `.c`, (in case you thought it twice and decided to use C instead of C++). We make some attempts before writing the command that does it.

```
; echo foo.cc | sed 's/.cc/.c/g'
foo.c
; f=foo.cc
; nf='{echo $f | sed 's/.cc/.c/g'}
; echo $nf
foo.c
; for (f in *.cc) {
;; nf='{echo $f | sed 's/.cc/.c/g'}
;; mv $f $nf
;; }
;       all of them renamed!
```

At this point, it should be easy for you to understand the command we used to generate the array initializer for hexadecimal numbers

```
sed -e 's/^/"0x/' -e 's/$/",/'
```

It had two editing commands, therefore we had to use `-e` for both ones. The first one replaced the start of a line with "`0x`", thus, it inserted this string at the beggining of line. The second inserted "`",`" at the end of line.

8.7. Moving files around

We want to copy all the files in a file tree to a single directory. Perhaps we have one directory per music album, and some files with songs inside.

```
; du -a
1       ./alanparsons/irobot.mp3
1       ./alanparsons/whatgoesup.mp3
2       ./alanparsons
1       ./pausini/trateilmare.mp3
1       ./pausini
1       ./supertramp/logical.mp3
1       ./supertramp
4       .
```

But we may want to burn a CD and we might need to keep the songs in a single directory. This can be done by using `cp` to copy each file of interest into another one at the target directory. But file names may not include `/`, and we want to preserve the album name. We can use `sed` to substitute the `/` with another character, and then copy the files.

```
; for (f in */*.mp3) {
;; nf='{echo $f | sed s,/,_,g}
;; echo cp $f /destdir/$nf
;; }
cp alanparsons/irobot.mp3 /destdir/alanparsons_irobot.mp3
cp alanparsons/whatgoesup.mp3 /destdir/alanparsons_whatgoesup.mp3
cp pausini/trateilmare.mp3 /destdir/pausini_trateilmare.mp3
cp supertramp/logical.mp3 /destdir/supertramp_logical.mp3
;
```

Here, we used a comma as the delimiter for the `sed` command, becase we wanted to use the slash in the expression to be replaced.

To copy the whole file tree to a different place, we cannot use `cp`. Even doing the same thing that we did above, we would have to create the directories to place the songs inside. That is a burden. A different strategy is to create an **archive** for the source tree, and then extract the archive at the destination. The command `tar`, was initially created to make tape archives. We no longer use tapes for achiving things. But `tar` remains a very useful command. A tape archive, also known as a tar-file, is a single file that contains many other ones (including directories)

bundled inside.

What `tar` does is to write to the beginning of the archive a table describing the file names and permissions, and where in the archive their contents start and terminate. This *header* is followed by the contents of the files themselves. The option `-c` creates one archive with the named files.

```
; tar -c * >/tmp/music.tar
```

We can see the contents of the archive using the option `t`.

```
; tar -t </tmp/music.tar
alanparsons/
alanparsons/irobot.mp3
alanparsons/whatgoesup.mp3
pausini/
pausini/trateilmare.mp3
supertramp/
supertramp/logical.mp3
```

Option `-v`, adds verbosity to the output, like in many other commands.

```
; tar -tv </tmp/music.tar
d-rwxr-xr-x        0 Jul 21 00:02 2006 alanparsons/
--rw-r--r--       13 Jul 21 00:01 2006 alanparsons/irobot.mp3
--rw-r--r--       13 Jul 21 00:02 2006 alanparsons/whatgoesup.mp3
d-rwxr-xr-x        0 Jul 21 00:02 2006 pausini/
--rw-r--r--       13 Jul 21 00:02 2006 pausini/trateilmare.mp3
d-rwxr-xr-x        0 Jul 21 00:02 2006 supertramp/
--rw-r--r--       13 Jul 21 00:02 2006 supertramp/logical.mp3
```

This lists the permissions and other file attributes. To extract the files in the archive, we can use the option `-x`. Here we add an `v` as well just to see what happens.

```
; cd otherdir
; tar xv </tmp/music.tar
alanparsons
alanparsons/irobot.mp3
alanparsons/whatgoesup.mp3
pausini
pausini/trateilmare.mp3
supertramp
supertramp/logical.mp3
; lc
alanparsons     pausini         supertramp
```

The size of the archive is a little bit more than the size of the files placed in it. That is to say that `tar` does not compress anything. If you want to compress the contents of an archive, so it occupies less space in the disk, you may use `gzip`. This is a program that uses a compression algorithm to exploit regularities in the data to use more efficient representation techiques for the same data.

```
; gzip music.tar
; ls -l music.*
--rw-r--r-- M 19 nemo nemo 10240 Jul 21 00:17 music.tar
--rw-r--r-- M 19 nemo nemo   304 Jul 21 00:22 music.tgz
```

The file `music.tgz` was created by `gzip`. In most cases, `gzip` adds the extension `.gz` for the compressed file name. But tradition says that compressed tar files terminate in `.tgz`.

Before extracting or inspecting the contents of a compressed archive, we must uncompress it. Below we also use the option `-f` for `tar`, that permits specifying the archive file as an

argument.

```
; tar -tf music.tgz
/386/bin/tar: partial block read from archive
; gunzip music.tgz
; tar -tf music.tar
alanparsons/
alanparsons/irobot.mp3
...etc...
```

So, how can we copy an entire file tree from one place to another? You now know how to use `tar`. Here is how.

```
; @{cd /music ; tar -c *}  | @{ cd /otherdir ; tar x }
```

The ouput for the first compound command goes to the input of the second one. The first one changes its directory to the source, and then creates an archive sent to standard output. In the second one, we change to the destination directory, and extract the archive read from standard input.

A new thing we have seen here is the expression `@{...}`, which is like `{...}`, but executes the command block in a child shell. We need to do this because each block must work at a different directory.