9 — More tools

9.1. Regular expressions

We have used `sed` to replace one string with another. But, what happens here?

```
; echo foo.xcc | sed 's/.cc/.c/g'
foo..c
; echo focca.x | sed 's/.cc/.c/g'
f.ca.x
```

We need to learn more.

In addresses of the form `/text/` and in commands like `s/text/other/`, the string `text` is not a string for `sed`. This happens to many other programs that search for things. For example, we have used `grep` to print only lines containing a string. Well, the *string* given to grep, like in

```
; grep string file1 file2 ...
```

is *not* a string. It is a **regular expression**. A regular expression is a little language. It is very useful to master it, because many commands employ regular expressions to let you do complex things in an easy way.

The text in a regular expression represents many different strings. You have already seen something similar. The `*.c` in the shell, used for globbing, is very similar to a regular expression. Although it has a slightly different meaning. But you know that in the shell, `*.c` **matches** with many different strings. In this case, those that are file names in the current directory that happen to terminate with the characters "`.c`". That is what regular expressions, or *regexps*, are for. They are used to select or match text, expressing the kind of text to be selected in a simple way. They are a language on their own. A regular expression, as known by `sed`, `grep`, and many others, is best defined recursively, as follows.

- Any single character *matches* the string consisting of that character. For example, `a` matches `a`, but not `b`.
- A single dot, "`.`", matches *any* single character. For example, "`.`" matches `a` and `b`, but not `ab`.
- A set of characters, specified by writing a string within brackets, like `[abc123]`, matches *any* character in the string. This example would match `a`, `b`, or `3`, but not `x`. A set of characters, but starting with `^`, matches any character *not* in the set. For example, `[^abc123]` matches `x`, but not `1`, which is in the string that follows the `^`. A range may be used, like in `[a-z0-9]`, which matches any single character that is a letter or a digit.
- A single `^`, matches the start of the text. And a single `$`, matches the end of the text. Depending on the program using the regexp, the text may be a line or a file. For example, when using `grep`, `a` matches the character `a` at *any* place. However, `^a` matches `a` only when it is the first character in a line, and `^a$` also requires it to be the last character in the line.
- Two regular expressions concatenated match any text matching the first regexp followed by any text matching the second. This is more hard to say than it is to understand. The expression `abc` matches `abc` because `a` matches `a`, `b` matches `b`, and so on. The expression `[a-z]x` matches any two characters where the first one matches `[a-z]`, and the second one is an `x`.
- Adding a `*` after a regular expression, matches zero or any number of strings that match the expression. For example, `x*` matches the empty string, and also `x`, `xx`, `xxx`, etc. Beware, `ab*` matches `a`, `ab`, `abb`, etc. But it does *not* match `abab`. The `*` applies to the preceding regexp, with is just `b` in this case.

- Adding a `+` after a regular expression, matches one or more strings that match the previous regexp. It is like `*`, but there has to be at least one match. For example, `x+` does not match the empty string, but it matches every other thing matched by `x*`.
- Adding a `?` after a regular expression, matches either the empty string or one string matching the expression. For example, `x?` matches `x` and the empty string. This is used to make parts optional.
- Different expressions may be surounded by parenthesis, to alter grouping. For example, `(ab)+` matches `ab`, `abab`, etc.
- Two expressions separated by `|` match anything matched either by the first, or the second regexp. For example, `ab|xy` matches `ab`, and `xy`.
- A backslash removes the special meaning for any character used for syntax. This is called a *escape* character. For example, `(` is not a well-formed regular expression, but `\(` is, and matches the string `(`. To use a backslash as a plain character, and not as a escape, use the backslash to escape itself, like in `\\`.

That was a long list, but it is easy to learn regular expressions just by using them. First, let's fix the ones we used in the last section. This is what happen to us.

```
; echo foo.xcc | sed 's/.cc/.c/g'
foo..c
; echo focca.x | sed 's/.cc/.c/g'
f.ca.x
```

But we wanted to replace `.cc`, and not *any* character and a `cc`. Now we know that the first argument to the `sed` command `s`, is a regular expression. We can try to fix our problem.

```
; echo foo.xcc | sed 's/\.cc/.c/g'
foo.xcc
; echo focca.x | sed 's/\.cc/.c/g'
focca.x
```

It seems to work. The backslash removes the special meaning for the dot, and makes it match just one dot. But this may still happen.

```
; echo foo.cc.x | sed 's/\.cc/.c/g'
foo.c.x
```

And we wanted to replace only the extension for file names ending in `.cc`. We can modify our expression to match `.cc` only when immediately before the end of the line (which is the string being matched here).

```
; echo foo.cc.x | sed 's/\.cc$/.c/g'
foo.cc.x
; echo foo.x.cc | sed 's/\.cc/.c/g'
foo.x.c
```

Sometimes, it is useful to be able to refer to text that matched part of a regular expression. Suppose you want to replace the variable name `text` with `word` in a program. You might try with `s/text/word/g`, but it would change other identifiers, which is not what you want.

```
; cat f.c
void
printtext(char* text)
{
        print("[%s]", text);
}
; sed 's/text/word/g' f.c
void
printword(char* word)
{
        print("[%s]", word);
}
```

The change is only to be done if `word` is not surounded by characters that may be part of an identifier in the program. For simplicity, we will assume that these characters are just `[a-z0-9_]`. We can do what follows.

```
; sed 's/([^a-z0-9_])text([^a-z0-9_])/\1word\2/g' f.c
void
printtext(char* word)
{
        print("[%s]", word);
}
```

The regular expression `[^a-z0-9_]text[^a-z0-9_]` means "any character that may not be part of an identifier", then `text`, and then "any character that may not be part of an identifier". Because the substitution affects *all* the regular expression, we need to substitute the matched string with another one that has `word` instead of `text`, but keeping the characters matching `[^a-z0-9_]` before and after the string `text`. This can be done by surounding in parentheses both `[^a-z0-9_]`. Later, in the destination string, we may use `\1` to refer to the text matching the first regexp within parenthesis, and `\2` to refer to the second.

Because `printtext` is not matched by `[^a-z0-9_]text[^a-z0-9_]`, it was untouched. However, "␣`text)`" was matched. In the destination string, `\1` was a white space, because that is what matched the first parenthesized part. And `\2` was a right parenthesis, because that is what matched the second one. As a result, we left those characters untouched, and used them as *context* to determine when to do the substitution.

Regular expressions permit to clean up source files in an easy way. In may cases, it makes no sense to keep white space at the end of lines. This removes them.

```
; sed 's/[ ]*//'
```

We saw that a script `t+` can be used to indent text in Acme. Here it is.

```
; cat /bin/t+
#!/bin/rc
sed 's/^/  /'
;
```

This other script removes one level of indentation.

```
; cat /bin/t-
#!/bin/rc
sed 's/^   //'
;
```

There are many other useful uses of regular expressions, as you will be able to see from here to the end of this book. In many cases, your C programs can be made more flexible by accepting regular expressions for certain parameters instead of mere strings. For example, an editor might accept a regular expression that determines if the text is to be shown using a `constant width`

`font` or a *proportional width font*. For file names matching, say `.*\.[ch]`, it could use a constant width font, to show C source code like you might be acustomed to see it.

It turns out that it is *trivial* to use regular expressions in a C program, by using the `regexp` library. The expression is *compiled* into a description more amenable to the machine, and the resulting data structure (called a `Reprog`) can be used for matching strings against the expression. This program accepts a regular expression as a parameter, and then reads one line at a time. For each such line, it reports if the string read matches the regular expression or not.

match.c

```
#include <u.h>
#include <libc.h>
#include <regexp.h>

void
main(int argc, char* argv[])
{
        Reprog* prog;
        Resub   sub[16];
        char    buf[1024];
        int     nr, ismatch, i;

        if (argc != 2){
                fprint(2, "usage: %s regexp\n", argv[0]);
                exits("usage");
        }
        prog = regcomp(argv[1]);
        if (prog == nil)
                sysfatal("regexp '%s': %r", buf);
        for(;;){
                nr = read(0, buf, sizeof(buf)-1);
                if (nr <= 0)
                        break;
                buf[nr] = 0;
                ismatch = regexec(prog, buf, sub, nelem(sub));
                if (!ismatch)
                        print("no match\n");
                else {
                        print("matched: '");
                        write(1, sub[0].sp, sub[0].ep - sub[0].sp);
                        print("'\n");
                }
        }
        exits(nil);
}
```

The call to `regcomp` *compiles* the regular expression into `prog`. Later, `regexec` *executes* the compiled regular expression to determine if it matches the string just read in `buf`. The parameter `sub` points to an array of structures that keeps information about the match. The whole string

matching starts at the character pointed to by `sub[0].sp` and terminates right before the one pointed to by `sub[0].ep`. Other entries in the array report which substring matched the first parenthesized expression in the regexp, `sub[1]`, which one matched the second one, `sub[2]`, etc. They are similar to `\1`, `\2`, etc. This is an example session with the program.

```
; 8.out '*.c'    The * needs something on the left!
regerror: missing operand for *

; 8.match '\.[123]'
x123
no match
.123
matched: '.1'
x.z
no match
x.3
matched: '.3'
```

9.2. Searching

We can revisit the first example in this chapter, finding function definitions. This script does just that, if we follow the style convention for declaring functions that was shown at the beginning of this chapter. First, we try to use `grep` to print just the source line where the function `cat` is defined in the file `/sys/src/cmd/cat.c`. Our first try is this.

```
; grep cat /sys/src/cmd/cat.c
cat(int f, char *s)
        argv0 = "cat";
                cat(0, "<stdin>");
                        cat(f, argv[i]);
```

Which is not too helpful. All the lines contain the string `cat`, but we want only the lines where `cat` is at the beggining of line, followed by an open parenthesis. Second attempt.

```
; grep '^cat\(' /sys/src/cmd/cat.c
cat(int f, char *s)
```

At least, this prints just the line of interest to us. However, it is useful to get the file name and line number before the text in the line. That output can be used to point an editor to that particular file and line number. Because `grep` prints the file name when more than one file is given, we could use `/dev/null` as a second file where to search for the line. It would not be there, but it would make `grep` print the file name.

```
; grep  '^cat\(' /sys/src/cmd/cat.c /dev/null
/sys/src/cmd/cat.c:cat(int f, char *s)
```

Giving the option `-n` to `grep` makes it print the line number. Now we can really search for functions, like we do next.

```
; grep -n '^cat\(' /sys/src/cmd/*.c
/sys/src/cmd/cat.c:5: cat(int f, char *s)
```

And because this seems useful, we can package it as a shell script. It accepts as arguments the names for functions to be located. The command `grep` is used to search for such functions at all the source files in the current directory.

```
#!/bin/rc
rfork e
for (f in $*)
        grep -n '^'$f'\('  *.[cCh]
```

How can we use `grep` to search for `-n`? If we try, `grep` would get confussed, thinking that we are supplying an option. To avoid this, the `-e` option tells `grep` that what follows is a regexp to search for.

```
; cat text
Hi there
How can we grep for -n?
Who knows!
; grep -n text
; grep -e -n text
how can we grep for -n?
```

There are other useful options. For example, if may want to locate lines in the file for a chapter of this book where we mention figures. However, if the word `figure` is in the middle of a sentence it would be all lowercase. When it is starting a sentece, it would be capitalized. We must search both for `Figure` and `figure.` The flag `-i` makes `grep` become case-insensitive. All the text read is converted to lowercase before matching the expression.

```
; grep -i figure ch1.ms
Each window shows a file or the output of commands.  Figure
figure are understood by acme itself. For commands
shown in the figure would be
```
...and other matching lines

A popular searching task is determining if a file containing a mail message is spam or not. Today, it would not work, because spammers employ heavy armoring, and even send their text encoded in multiple images sent as HTML mail. However, it was popular to see if a mail message contained certain expressions, if it did, it was considered spam. Because there will be many expressions, we may keep them in a file. The option `-f` for `grep` takes as an argument a file containing all the expressions to search for.

```
; cat patterns
Make money fast!
Earn 10+ millions
(Take|use) viagra for a (better|best) life.
; if (grep -i -f patterns $mailfile ) echo $mailfile is spam
```

Show diff, to search for differences. See the script bdiff

9.3. AWK

A simple calculator. Pickup some examples from your bin

9.4. More complex things

Show the scripts to replace strings. Show the script to automatically add users from a list of people.

Show eval. Use walk as an example. Show doctype also.

must talk about other file systems, plumber, cdfs, tarfs, upasfs, the dump