CS 114 – Fall 2004

Lecture 7 – Wednesday, October 13, 2004

Text processing

sed

Last time, we introduced the `sed`, which can be used to substitute a string for a string matching a regular expression. For example, the following `sed` command substitutes the empty string "" for any spaces at the beginning of the line in the input.

```
% sed 's/^ *//'
```

If you just run this command on it's own, `sed` will block waiting for input. If you type a few lines of text, `sed` will process each line as it is entered. You can tell `sed` (or any program expecting input from `stdin`) that there is no more input by typing `Ctrl-d`. This signifies *end of file*. `sed` will stop reading input and exit. You can also give `sed` a file as input by doing:

```
% sed 's/^ *//' < file
```

or just:

```
% sed 's/^ *//' file
```

As `sed` commands become more complex, you will want to put them in a separate file for easier editing and reuse. For example, a `sed` script to remove both leading and trailing spaces could be put in a file, `trim.sed`:

```
% cat trim.sed
s/^ *//
s/ *$//
```

The script can be run on the input file `file` with the `sed` command below:

```
% cat file
   hello, world!
% sed -f trim.sed < file
hello, world!
```

The `sed` script can be made into a *program* by first making the script executable with `chmod`:

```
% chmod +x trim.sed
```

Then, adding the following line to the beginning of the file:

```
#!/usr/bin/sed -f
```

The script now looks like this:

```
% cat trim.sed
#!/usr/bin/sed -f
s/^ *//
s/ *$//
```

The #! is called a *hash bang* or *sh-bang*. When an executable file begins with #!, the operating system reads the rest of the line after the #! and invokes that command, passing the file as an additional argument. Thus, if you run:

```
% ./trim.sed file
```

The operating system will invoke the command:

```
/usr/bin/sed -f ./trim.sed file
```

Note that the full path to sed must be given on the #! line because the operating system, not the shell interprets the command. The PATH is not searched and wildcards are not expanded.

A non-trivial shell script

The following sed script centers a line of text on 80 columns (the standard width of a terminal).

```
% cat center
#!/usr/bin/sed -f
# Center all lines of a file on 80 columns width.
# To change that width, the number in \{\} must be replaced,
# and the number of added spaces also must be changed

# Replace tabs with spaces: This first line has a tab in the first
# place and a space in the second place
y/      / /

# Delete leading and trailing spaces.
s/^ *//
s/ *$//

# Add 80 spaces to end of line.
# The following line has 10 spaces in the second place
s/$/          /
# This line replaces the 10 spaces at the end of each line
# with 8 copies of those spaces; "&" is the string matched by
# the regular expression.
s/ *$/&&&&&&&&/

# Keep 1st 80 chars.
s/^\(.\{80\}\).*$/\1/

# Split trailing spaces in half and move the first half up front.
# The trailing " *" is there to handle the case of an odd number of
# trailing spaces: " *" should match only zero or one space
s/^\(.*[^ ]\)\( *\)\2 *$/\2\1/
```

sed runs each command in the script once per input line, in order. For each line, the script works by first replacing tabs with spaces using the y/// command. Next, it removes leading and trailing whitespace. It then appends 80 spaces and then truncates the line to 80 columns. Finally, it uses the grouping operators \(\) to capture the non-spaces on the line in \1 and to capture the first half of the trailing spaces in \2. The second half is matched by \2 in the regular expression. Since there may be an odd number of trailing spaces, the last (optional) space is matched by " *" at the end of the regular expression. This line is replaced with "\2\1": that is, the first half of the spaces captured in \2 and the non-spaces in \1.

AWK

AWK is another text processing language found on Unix systems. An AWK file consists of three parts: an optional `BEGIN` block, zero or more guarded blocks, and an optional `END` block. An AWK file thus looks something like this:

```
BEGIN { ... }
guard1 { ... }
guard2 { ... }
...
guardn { ... }
END { ... }
```

The `BEGIN` block is run before processing any input. The `END` block is run after processing all input. The guarded blocks are run for each line of the input for which the guard is true. A missing guard evaluates to true.

For example, the following AWK script prints the second column (separated by whitespace) of each line of input that matches the regular expression "`^ *cs114`"

```
/^ *cs114/ { print $2 }
```

This script can be invoked directly on the command line as:

```
% awk '/^ *cs114/ { print $2 }' inputfile
```

It can also be invoked by saving the script to a file and running `awk -f`, or by adding `#!/usr/bin/awk -f` to the script file and making it executable.

To change the column separator, you can set the `FS` (field separator) variable in the begin block. For example, the following AWK command prints the username and shell of the user named `root` in /etc/passwd:

```
% grep ^root /etc/passwd
root:*:0:0:System Administrator:/var/root:/bin/sh
% awk 'BEGIN { FS = ":" } $1 ~ /root/ && NF == 7 { print $1 " " $7 }'
/etc/passwd
root /bin/sh
```

AWK can also be used to center lines of text, much more straightforwardly than `sed`. For example:

```
% cat center.awk
#!/usr/bin/awk -f
{ printf "%" int(40+length($0)/2) "s\n", $0 }
```