Editor: Diomidis Spinellis ■ Athens University of Economics and Business ■ dds@aueb.gr

Working with Unix Tools

Diomidis Spinellis

A successful [software] tool is one that was used to do something undreamed of by its author.
— Stephen C. Johnson

Line-oriented textual data streams are the lowest useful common denominator for a lot of data that passes through our hands. We use such streams to represent program source code, Web server log data, version control history, file lists, symbol tables, archive contents, error messages, profiling data, and so on. For many routine, everyday tasks, we might be tempted to process the data using a "Swiss army knife" scripting language such as Perl, Python, or Ruby. However, doing that often requires writing a small, self-contained program and saving it into a file. By that point, we've sometimes lost interest in the task and end up doing the work manually, if at all. Often, it's more effective to combine Unix toolchest programs into a short and sweet pipeline that we can run from our shell's command prompt. With modern shell command-line editing facilities, we can build our command bit by bit, until it molds into exactly the form that suits us. Nowadays, many different systems—including GNU/Linux, Mac OS X, and Microsoft Windows—offer the original Unix tools preinstalled or as free downloads, so there's no excuse for not adding this approach to your arsenal.

Many one-liners that you'll build around the Unix tools follow a pattern that goes roughly like this: data fetching, selection, processing, and summarization. You'll also need to apply some plumbing to these parts. Jump in to get a quick tour of the facilities.

Getting the data

Most of the time, your data will be text that you can feed directly to a tool's standard input. If this isn't the case, you'll need to adapt your data. If you're dealing with object files, try a command like `nm` (Unix), `dumpbin` (Windows), or `javap` (Java) to dig into them. If you're working with files grouped into an archive, a command like `tar`, `jar`, or `ar` will list the archive's contents. If your data comes from a (potentially large) collection of files, `find` can locate those that interest you. Then again, to get your data over the Web, use `wget`. You can also use `dd` (and the special file /dev/zero), `yes`, or `jot` to generate artificial data, perhaps to run a quick benchmark. Finally, to process a compiler's list of error messages, redirect its standard error to its standard output; the incantation `2>&1` will do the trick.

I've not covered many other cases, including relational databases, version control systems, mail clients, office applications, and so on. Keep in mind that you're unlikely to be the first to need the application's data converted to a textual format, so the tool you need probably already exists. For example, my Outwit tool suite (www.spinellis.gr/sw/outwit) can convert into a text stream data coming from the Windows clipboard, an ODBC (open database connectivity) source, the event log, or the registry.

Selection

Given the textual data format's generality, you'll frequently have more data than you need. You might want to process only some parts of

each row or only a subset of the rows. To select a specific column from a line consisting of elements separated by spaces or another field delimiter, use `awk` with a single `print $n` command. If your fields have a fixed width, you can separate them using `cut`. And, if your lines are not neatly separated into fields, you might write a regular expression for a `sed` substitute command to isolate the desired element.

The workhorse for obtaining a subset of the rows is `grep`. Specify a regular expression to get only the rows that match it and add the `-v` flag to filter out rows you don't want to process. Use `fgrep` with the `-f` flag if the elements you're looking for are fixed and stored in a file (perhaps generated in a previous processing step). If your selection criteria are more complex, you might express them in an `awk` pattern expression. Many times you'll find yourself combining several of these approaches. For example, you might use `grep` to get the lines that interest you, `grep -v` to filter out some noise from your sample, and finally `awk` to select a specific field from each line.

Processing

You'll find that data processing frequently involves sorting your lines on a specific field. The `sort` command supports tens of options for specifying the sort keys, their type, and the output order. Having your results sorted, you could then count how many instances of each element you have. The `uniq` command with the `-c` (count) option will do the job here; you might also postprocess the result with another `sort`, this time with the `-n` flag specifying a numerical order, to find out which elements appear most frequently. In other cases, you could compare results between different runs. You can use `diff` if the two runs generate results that should be the same (perhaps a regression test's) or `comm` if you want to compare two sorted lists. Again, you'll handle more complex tasks using `awk`.

All the wonderful building blocks we've described are useless without some way to glue them together.

Summarizing

In many cases, the processed data is too voluminous to be useful. For example, you might not care which symbols are defined with the wrong visibility in your program, but you might want to know how many exist. Surprisingly, many problems involve simply counting the processing step's output using the humble `wc` (word count) command and its `-l` (count lines) flag. If you want to know the top or bottom 10 elements of your results list, you can pass your list through `head` or `tail`. To format a long list of words into a more manageable block that you can paste into your code, use `fmt` (perhaps run after a `sed` substitution command tacks a comma after each element). Also, for debugging purposes, you might initially pipe the result of intermediate stages through `more` or `less` to examine it in detail. As usual, use `awk` when these approaches don't suit you; a typical task involves summing up a specific field with a command such as `sum += $3`.

Plumbing

All the wonderful building blocks we've described are useless without some way to glue them together. For this, you'll use the Bourne shell's facilities. First and foremost comes the pipeline (`|`), which lets you send one processing step's output as input to the next one. In other cases, you might want to execute the same command with many different arguments. For this, you'll pass the arguments as input to `xargs`. A typical pattern involves obtaining a list of files using `find` and processing them using `xargs`. So common is this pattern that, to handle files with embedded spaces in them, both commands nowadays support an argument (`-print0` and `-0`) to have their data terminated with a null character instead of a space. If your processing is more complex, you can always pipe the arguments into a `while read` loop (amazingly, the Bourne shell lets you pipe data to and from all its control structures). When all else fails, don't shy away from using a couple of intermediate files to juggle your data.

Putting it all together

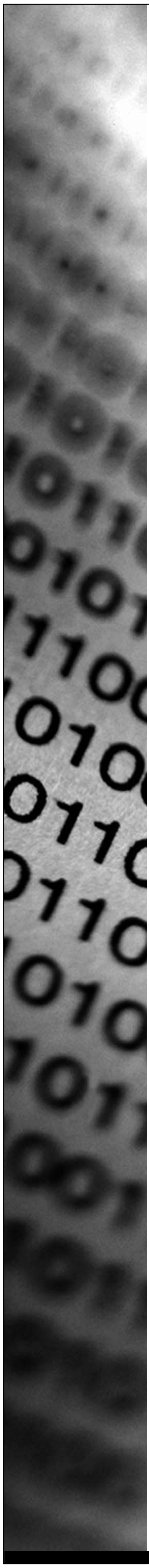
The following command will examine all Java files located in the directory `src` and print the 10 files with the highest number of occurrences of a method call to `substring`:

```
find src -name '*.java' -print |
xargs fgrep -c .substring |
sort -t: -rn -k2 |
head -10
```

The pipeline sequence will first use `find` to locate the Java files and apply `fgrep` to them, counting (`-c`) the occurrences of `.substring`. Then, `sort` will order the results in reverse numerical order (`-rn`) according to the second field (`-k2`) using `:` as the separator (`-t:`), and `head` will print the top 10 files.

Appalled? Confused? Disheartened? Don't worry. It took me four iterations and two manual lookups to get the above command exactly right, but it was still much faster than counting by hand or writing a program to do the counting. Every time you concoct a pipeline, you become a little better at it. Before you know it, you'll become the hero of your group: the one who knows the commands that can do magic.

Diomidis Spinellis is an associate professor in the Department of Management Science and Technology at the Athens University of Economics and Business and the author of *Code Reading: The Open Source Perspective* (Addison-Wesley, 2003). Contact him at dds@aueb.gr.



SENIOR MEMBERS TECHNICAL STAFF

The Software Engineering Institute (SEI) is a federally funded research and development center and a college-level unit of Carnegie Mellon University. The sustained growth of our Software Architecture Technology (SAT) Initiative has created the following opportunities:

Software Architecture Research

Position #1201

Working with SEI software architecture experts, selected candidates will conduct research in the areas of architecture design, reconstruction, and/or self-healing architectures; participate in the development of an expert architectural design assistant; and author publication-quality technical reports.

This position requires a PhD in a software engineering related field; 5+ years of direct system experience; relevant technical publications; demonstrated skill in real-time performance, reliability, security, or other quality attributes; and technical currency in leading-edge software technology.

Software Architecture Life Cycle Integration

Position #1205

Providing technical leadership in the development and transition of software architecture methods across the life-cycle, selected candidate will participate in the development and integration of architecture-centric life cycle practices; deliver courses in software architecture; and author publication-quality technical reports.

This position requires an M.S. in a software engineering related field; 8+ years of direct systems experience as a practitioner and/or educator in software engineering; technical currency in leading-edge software technology.

We offer excellent compensation, comprehensive benefits and the opportunity to work in a collegial results-oriented environment. Candidates interested in joining our team should go to **http://hr.web.cmu.edu/prospective** to create an applicant profile and apply online. AA/EOE

www.sei.cmu.edu

Carnegie Mellon
Software Engineering Institute