Structural Regular Expressions

Rob Pike

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The current UNIX® text processing tools are weakened by the built-in concept of a line. There is a simple notation that can describe the 'shape' of files when the typical array-of-lines picture is inadequate. That notation is regular expressions. Using regular expressions to describe the structure in addition to the contents of files has interesting applications, and yields elegant methods for dealing with some problems the current tools handle clumsily. When operations using these expressions are composed, the result is reminiscent of shell pipelines.

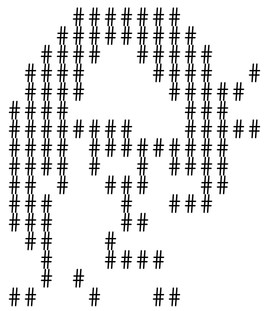
The Peter-On-Silicon Problem

In the traditional model, UNIX text files are arrays of lines, and all the familiar tools — `grep`, `sort`, `awk`, etc. — expect arrays of lines as input. The output of `ls` (regardless of options) is a list of files, one per line, that may be selected by tools such as `grep`:

```
ls -l /usr/ken/bin | grep 'rws.*root'
```

(I assume that the reader is familiar with the UNIX tools.) The model is powerful, but it is also pervasive, sometimes overly so. Many UNIX programs would be more general, and more useful, if they could be applied to arbitrarily structured input. For example, `diff` could in principle report differences at the C function level instead of the line level. But if the interesting quantum of information isn't a line, most of the tools (including `diff`) don't help, or at best do poorly. Worse, perverting the solution so the line-oriented tools can implement it often obscures the original problem.

To see how a line oriented view of text can introduce complication, consider the problem of turning Peter into silicon. The input is an array of blank and non-blank characters, like this:



The output is to be statements in a language for laying out integrated circuits:

```
rect minx miny maxx maxy
```

The statements encode where the non-blank characters are in the input. To simplify the problem slightly, the coordinate system has *x* positive to the right and *y* positive down. The output need not be efficient in its use of rectangles. `Awk` is the obvious language for the task, which is a mixture of text processing and geometry, hence arithmetic. Since the input is an array of lines, as `awk` expects, the job should be fairly easy, and in fact it is. Here is an awk program for the job:

```
BEGIN{
     y=1
}
/^/{
     for(x=1; x<=length($0); x++)
          if(substr($0, x, 1)=="#")
               print "rect", x, y, x+1, y+1
     y++
}
```

Although it is certainly easy to write, there is something odd about this program: the line-driven nature of awk results in only one obvious advantage — the ease of tracking y. The task of breaking out the pieces of the line is left to explicit code, simple procedural code that does not use any advanced technology such as regular expressions for string manipulation. This peculiarity becomes more evident if the problem is rephrased to demand that each horizontal run of rectangles be folded into a single rectangle:

```
BEGIN{
     y=1
}
/^/{
     for(x=1; x<=length($0); x++)
          if(substr($0, x, 1)=="#"){
               x0=x;
               while(++x<=length($0) && substr($0, x, 1)=="#")
                    ;
               print "rect", x0, y, x, y+1
          }
     y++
}
```

Here a considerable amount of code is being spent to do a job a regular expression could do very simply. In fact, the only regular expression in the program is ^, which is almost irrelevant to the input. (Newer versions of awk have mechanisms to use regular expressions within actions, but even there the relationship between the patterns that match text and the actions that manipulate the text is still too weak.)

Awk's patterns — the text in slashes // that select the input on which to run the actions, the programs in the braces {} — pass to the actions the entire line containing the text matched by the pattern. But much of the power of this idea is being wasted, since the matched text can only be a line. Imagine that awk were changed so the patterns instead passed precisely the text they matched, with no implicit line boundaries. Our first program could then be written:

```
BEGIN{
     x=1
     y=1
}
/ /{
     x++
}
/#/{
     print "rect", x, x+1, y, y+1
     x++
}
/\n/{
     x=1
     y++
}
```

and the second version could use regular expressions to break out complete strings of blanks and #'s simply:

```
BEGIN{
     x=1
     y=1
}
/ +/{
     x+=length($0)
}
/#+/{
     print "rect", x, x+length($0), y, y+1
     x+=length($0)
}
/\n/{
     x=1
     y++
}
```

In these programs, regular expressions are being used to do more than just select the input, the way they are used in all the traditional UNIX tools. Instead, the expressions are doing a simple parsing (or at least a breaking into lexical tokens) of the input. Such expressions are called *structural regular expressions* or just *structural expressions.*

These programs are not notably shorter than the originals, but they are conceptually simpler, because the structure of the input is expressed in the structure of the programs, rather than in procedural code. The labor has been cleanly divided between the patterns and the actions: the patterns select portions of the input while the actions operate on them. The actions contain no code to disassemble the input.

The lexical analysis generator `lex` uses regular expressions to define the structure of text, but its implementation is poor, and since it is not an interactive program (its output must be run through the C compiler) it has largely been forgotten as a day-to-day tool. But even ignoring issues of speed and convenience, `lex` still misses out on one of the most important aspects of structural expressions. As the next section illustrates, structural expressions can be nested to describe the structure of a file recursively, with surprising results.

Interactive Text Editing

It is ironic that UNIX files are uninterpreted byte streams, yet the style of programming that most typifies UNIX has a fairly rigid structure imposed on files — arrays of not-too-long lines. (The silent limits placed on line lengths by most tools can be frustrating.) Although the `awk` variant introduced above does not exist, there is an interactive text editor, `sam`, that treats its files as simple byte streams.

The sam command language looks much like that of ed, but the details are different because sam is not line-oriented. For example, the simple address

```
/string/
```

matches the next occurrence of ‘‘string’’, not the next line containing ‘‘string’’. Although there are shorthands to simplify common actions, the idea of a line must be stated explicitly in sam.

Sam has the same simple text addition and modification commands ed has: a adds text after the current location, i adds text before it, d deletes it, and c replaces it.

Unlike in ed, the current location in sam need not be (and usually isn’t) a line. This simplifies some operations considerably. For example, ed has several ways to delete all occurrences of a string in a file. One method is

```
g/string/ s///g
```

It is symptomatic that a substitute command is used to delete text within a line, while a delete command is used to delete whole lines. Also, if the string to be deleted contains a newline, this technique doesn’t work. (A file is just an array of characters, but some characters are more equal than others.) Sam is more forthright:

```
x/string/d
```

The x (‘extract’) command searches for each occurrence of the pattern, and runs the subsequent command with the current text set to the match (not to the line containing the match). Note that this is subtly different from ed’s g command: x extracts the complete text for the command, g merely selects lines. There is also a complement to x, called y, that extracts the pieces *between* the matches of the pattern.

The x command is a loop, and sam has a corresponding conditional command, called g (unrelated to ed’s g):

```
g/pattern/command
```

runs the command if the current text matches the pattern. Note that it does not loop, and it does not change the current text; it merely selects whether a command will run. Hence the command to print all lines containing a string is

```
x/.*\n/ g/string/p
```

— extract all the lines, and print each one that contains the string. The reverse conditional is v, so to print all lines containing ‘rob’ but not ‘robot’:

```
x/.*\n/ g/rob/ v/robot/p
```

A more dramatic example is to capitalize all occurrences of words ‘i’:

```
x/[A-Za-z]+/ g/i/ v/../ c/I/
```

— extract all the words, find those that contain ‘i’, reject those with two or more characters, and change the string to ‘I’ (borrowing a little syntax from the substitute command). Some people have overcome the difficulty of selecting words or identifiers using regular expressions by adding notation to the expressions, which has the disadvantage that the precise definition of ‘identifier’ is immutable in the implementation. With sam, the definition is part of the program and easy to change, although more long-winded.

The program to capitalize ‘i’s should be writable as

```
x/[A-Za-z]+/ g/^i$/ c/I/
```

That is, the definition of ^ and $ should reflect the structure of the input. For compatibility and because of some problems in the implementation, however, ^ and $ in sam always match line boundaries.

In ed, it would not be very useful to nest global commands because the ‘output’ of each global is still a line. However, sam’s extract commands can be nested effectively. (This benefit comes from separating the notions of looping and matching.) Consider the problem of changing all occurrences of the variable n in a C program to some other name, say num. The method above will work —

```
x/[a-zA-Z0-9]+/ g/n/ v/../ c/num/
```

— except that there are places in C where the 'identifier' `n` occurs but not as a variable, in particular as the constant `\n` in characters or strings. To prevent incorrect changes, the command can be prefixed by a couple of `y` commands to weed out characters and strings:

```
y/".*"/ y/'.*'/ x/[a-zA-Z0-9]+/ g/n/ v/../ c/num/
```

This example illustrates the power of composing extractions and conditionals, but it is not artificial: it was encountered when editing a real program (in fact, `sam`). There is an obvious analogy with shell pipelines, but these command *chains* are subtly — and importantly — different from pipelines. Data flows into the left end of a pipeline and emerges transformed from the right end. In chains, the data flow is implicit: all the commands are operating on the same data (except that the last element of the chain may modify the text); the complete operation is done in place; and no data actually flows through the chain. What is being passed from link to link in the chain is a view of the data, until it looks right for the final command in the chain. The data stays the same, only the structure is modified.

More than one line, and less than one line

The standard UNIX tools have difficulty handling several lines at a time, if they can do so at all. `Grep`, `sort` and `diff` work on lines only, although it would be useful if they could operate on larger pieces, such as a `refer` database. `awk` can be tricked into accepting multiple-line records, but then the actions must break out the sub-pieces (typically ordinary lines) by explicit code. `sed` has a unique and clumsy mechanism for manipulating multiple lines, which few have mastered.

Structural expressions make it easy to specify multiple-line actions. Consider a `refer` database, which has multi-line records separated by blank lines. Each line of a record begins with a percent sign and a character indicating the type of information on the line: `A` for author, `T` for title, etc. Staying with `sam` notation, the command to search a `refer` database for all papers written by Bimmler is:

```
x/(.+\n)+/ g/%A.*Bimmler/p
```

— break the file into non-empty sequences of non-empty lines and print any set of lines containing 'Bimmler' on a line after '%A'. (To be compatible with the other tools, a '`.`' does not match a newline.) Except for the structural expression, this is a regular `grep` operation, implying that `grep` could benefit from an additional regular expression to define the structure of its input. In the short term, however, a 'stream `sam`,' analogous to `sed`, would be convenient, and is currently being implemented.

The ability to compose expressions makes it easy to tune the search program. For example, we can select just the *titles* of the papers written by Bimmler by applying another extraction:

```
x/(.+\n)+/ g/%A.*Bimmler/ x/.*\n/ g/%T/p
```

This program breaks the records with author Bimmler back into individual lines, then prints the lines containing `%T`.

There are many other examples of multiple-line components of files that may profitably be extracted, such as C functions, messages in mail boxes, paragraphs in `troff` input and records in on-line telephone books. Note that, unlike in systems that define file structures *a priori*, the structures are applied by the program, not the data. This means the structure can change from application to application; sometimes a C program is an array of functions, but sometimes it is an array of lines, and sometimes it is just a byte stream.

If the standard commands admitted a structural expression to determine the appearance of their input, many currently annoying problems could become simple: imagine a version of `diff` that could print changed sentences or functions instead of changed lines, or a `sort` that could sort a `refer` database. The case of `sort` is particularly interesting: not only can the shape of the input records be described by a structural expression, but also the shape of the sort key. The current bewildering maze of options to control the sort could in principle be largely replaced by a structural expression to extract the key from the record, with multiple expressions to define multiple keys.

The `awk` of the future?

It is entertaining to imagine a version of `awk` that applies these ideas throughout. First, as discussed earlier, the text passed to the actions would be defined, rather than merely selected, by the patterns. For example,

```
/#+/ { print }
```

would print only # characters; conventional `awk` would instead print every line containing # characters.

Next, the expressions would define how the input is parsed. Instead of using the restrictive idea of a field separator, the iterations implied by closures in the expression can demarcate fields. For instance, in the program

```
/(.+\n)+/ { action }
```

the action sees groups of lines, but the outermost closure (the + operator) examines, and hence can extract, the individual lines. `ed` uses parentheses to define sub-expressions for its back-referencing operators. We can modify this idea to define the 'fields' in `awk`, so `$1` defines the first element of the closure (the first line), `$2` the second, and so on. More interestingly, the closures could generate indices for arrays, so the fields would be called, say, `input[1]` and so on, perhaps with the unadorned identifier `input` holding the original intact string. This has the advantage that nested closures can generate multi-dimensional arrays, which is notationally clean. (There is some subtlety involving the relationship between `input` indices and the order of the closures in the pattern, but the details are not important here.)

Finally, as in `sam`, structural expressions would be applicable to the output of structural expressions; that is, we would be able to nest structural expressions inside the actions. The following program computes how many pages of articles Bimmler has written:

```
/(.+\n)+/{    # break into records
     input ~ /%A.*Bimmler/{    # is Bimmler author? (see text)
          /%P.*([0-9]+)-([0-9]+)/{    # extract page numbers
               pages+=input[2]-input[1]+1
          }
     }
}
END{
     print pages
}
```

Real `awk` uses patterns (that is, regular expressions) only like `sam`'s `g` command, but our `awk`'s patterns are `x` expressions. Obviously, we need both to exploit structural expressions well. This is why in the program above the test for whether `input` contains a paper by Bimmler must be written as an explicit pattern match. The innermost pattern searches for lines containing two numbers separated by a dash, which is how `refer` stores the starting and ending pages of the article.

This is a contrived example, of course, but it illustrates the basic ideas. The real `awk` suffers from a mismatch between the patterns and the actions. It would be improved by making the parsing actions of the patterns visible in the actions, and by having the pattern-matching abilities available in the actions. A language with regular expressions should not base its text manipulation on a `substr` function.

Comments

The use of regular expressions to describe the structure of files is a powerful and convenient, if unfamiliar, way to address a number of difficulties the current UNIX tools share. There is obviously around this new notation a number of interesting problems, and I am not pretending to have addressed them all. Rather, I have skipped enthusiastically from example to example to indicate the breadth of the possibilities, not the depth of the difficulties. My hope is to encourage others to think about these ideas, and perhaps to apply them to old tools as well as new ones.

Acknowledgements

John Linderman, Chris Van Wyk, Tom Duff and Norman Wilson will recognize some of their ideas in these notes. I hope I have not misrepresented them.