A Regular Expression Matcher

Code by Rob Pike

Exegesis by Brian Kernighan

Draft version Jan 28 2007

Introduction

Beautiful code is likely to be simple -- clear and easy to understand. Beautitful code is likely to be compact -- just enough code to do the job and no more -- but not cryptic, to the point where it cannot be understood. Beautiful code may well be general, solving a broad class of problems in a uniform way. One might even describe it as elegant, showing good taste and refinement.

In this chapter I will describe a piece of beautiful code, for matching regular expressions that meets all of these criteria.

Regular expressions are a notation for describing patterns of text, in effect a special-purpose language for pattern matching. Although there are many variants, all share the idea that most characters in a pattern match literal occurrences of themselves but some "metacharacters" have special meaning, for example "`*`" to indicate some kind of repetition or `[...]` to mean any one character from the set within the brackets.

In practice, most searches in programs like text editors are for literal words, so the regular expressions are often literal strings like `print`, which will match `printf` or `sprint` or `printer paper` anywhere. In so-called "wild cards" in filenames in Unix and Windows, a `*` matches any number of characters, so the pattern `*.c` matches all filenames that end in `.c`. There are many, many variants of regular expressions, even in contexts where one would expect them to be the same. Jeffrey Friedl's *Mastering Regular Expressions* (O'Reilly, 2006) is an exhaustive study of the topic.

Regular expressions were invented by Stephen Kleene in the mid-1950's as a notation for finite automata, and in fact they are equivalent to finite automata in what they represent. Regular expressions first appeared in a program setting in Ken Thompson's version of the QED text editor in the mid-1960's. In 1967, Ken applied for a patent on a mechanism for rapid text matching based on regular expressions; it was granted in 1971, one of the very first software patents [US Patent 3,568,156, Text Matching Algorithm, March 2, 1971].

Regular expressions moved from QED to the Unix editor ed, and then to the quintessential Unix tool, grep, which Ken created by performing radical surgery on ed. Through these widely used programs, regular expressions became familiar throughout the early Unix community.

Ken's original matcher was very fast because it combined two independent ideas. One was to generate machine instructions on the fly during matching so that it ran at machine speed, not by interpretation. The other was to carry forward all possible matches at each stage, so it did not have to backtrack to look for alternative potential matches. Matching code in later text editors that Ken wrote, like ed, used a simpler algorithm that backtracked when necessary. In theory this is slower but the patterns found in practice rarely involved backtracking, so the ed and grep algorithm and code were good enough for most purposes.

Subsequent regular expression matchers like egrep and fgrep added richer classes of regular expressions, and focused on fast execution no matter what the pattern. Ever fancier regular expressions became popular, and were included not only in C-based libraries but also as part of the syntax of scripting languages like Awk and Perl.

The Practice of Programming

In 1998, Rob Pike and I were writing *The Practice of Programming* ("TPOP"). The last chapter of the book, "Notation", collected a number of examples where a good notation led to better programs and better programming. This included the use of simple data specifications (`printf` formats, for instance), and the generation of code from tables.

Given our Unix backgrounds and many years of experience with tools based on regular expression notation, we naturally wanted to include a discussion of regular expressions, and it seemed mandatory to include an implementation as well. Given our emphasis on tools, it also seemed best to focus on the class of regular expressions found in grep, rather than say those from shell wild cards, since we could also talk about the design of grep itself.

The problem was that any existing regular expression package was far too big. The local grep was over 500 lines long (about 10 book pages). Open-source regular expression packages tended to be huge, roughly the size of the entire book, because they were engineered for generality, flexibility, and speed; none was remotely suitable for pedagogy.

I suggested to Rob that we needed to find the smallest regular expression package that would illustrate the basic ideas while still recognizing a useful and non-trivial class of patterns. Ideally, the code would fit on a single page.

Rob disappeared into his office, and at least as I remember it now, appeared again in no more than an hour or two with the 30 lines of C code that subsequently appeared in Chapter 9 of TPOP. That code implements a regular expression matcher that handles these constructs:

```
c    matches any literal character c
.    matches any single character
^    matches the beginning of the input string
$    matches the end of the input string
*    matches zero or more occurrences of the previous character
```

This is quite a useful class; in my own experience of using regular expressions on a day-to-day basis, it easily accounts for 95 percent of all instances. In many situations, solving the right problem is a big step on the road to a beautiful program. Rob deserves great credit for choosing so wisely, from among a wide set of options, a very small yet important, well-defined and extensible set of features.

Rob's implementation itself is a superb example of beautiful code: compact, elegant, efficient, and useful. It's one of the best examples of recursion that I have ever seen, and it shows the power of C pointers. Although at the time we were most interested in conveying the important role of a good notation in making a program easier to use and perhaps easier to write as well, the regular expression code has also been an excellent way to illustrate algorithms, data structures, testing, performance enhancement, and other important topics.

Implementation

In the book, the regular expression matcher is part of a program that mimics grep, but the regular expression code is completely separable from its surroundings. The main program is not interesting here -- it simply reads from its standard input or from a sequence of files, and prints those lines that contain a match of the regular expression, as does the original grep and many other Unix tools.

This is the matching code:

```
/* match: search for regexp anywhere in text */
int match(char *regexp, char *text)
{
    if (regexp[0] == '^')
        return matchhere(regexp+1, text);
    do {    /* must look even if string is empty */
        if (matchhere(regexp, text))
            return 1;
```

Inner: 1798 x 984
Outer: 1798 x 1058

```
    } while (*text++ != '\0');
    return 0;
}

/* matchhere: search for regexp at beginning of text */
int matchhere(char *regexp, char *text)
{
    if (regexp[0] == '\0')
        return 1;
    if (regexp[1] == '*')
        return matchstar(regexp[0], regexp+2, text);
    if (regexp[0] == '$' && regexp[1] == '\0')
        return *text == '\0';
    if (*text!='\0' && (regexp[0]=='.' || regexp[0]==*text))
        return matchhere(regexp+1, text+1);
    return 0;
}

/* matchstar: search for c*regexp at beginning of text */
int matchstar(int c, char *regexp, char *text)
{
    do {    /* a * matches zero or more instances */
        if (matchhere(regexp, text))
            return 1;
    } while (*text != '\0' && (*text++ == c || c == '.'));
    return 0;
}
```

The function `match(regexp, text)` tests whether there is an occurrence of the regular expression anywhere within the text; it returns 1 if a match is found and 0 if not. If there is more than one match, it finds the leftmost and shortest.

The basic operation of `match` is straightforward. If the first character of the regular expression is `^` (an anchored match), any possible match must occur at the beginning of the string. That is, if the regular expression is `^xyz`, it matches `xyz` only if `xyz` occurs at the beginning of the text, not somewhere in the middle. This is tested by matching the rest of the regular expression against the text starting at the beginning, and nowhere else.

Otherwise, the regular expression might match anywhere within the string. This is tested by matching the pattern against each character position of the text in turn. If there are multiple matches, only the first (leftmost) one will be identified. That is, if the regular expression is `xyz`, it will match the first occurrence of `xyz` regardless of where it occurs.

Notice that advancing over the input string is done with a `do-while` loop, a comparatively unusual construct in C programs. The occurrence of a `do-while` instead of a `while` should always raise a question: why isn't the loop termination condition being tested at the beginning of the loop, before it's too late, rather than at the end after something has been done? But the test is correct here: since the `*` operator permits zero-length matches, we first have to check whether a null match is possible.

The bulk of the work is done in the function `matchhere(regexp, text)`, which tests whether the regular expression matches the text that begins right here. The function `matchhere` operates by attempting to match the first character of the regular expression with the first character of the text. If the match fails, there can be no match at this text position and `matchhere` returns 0. If the match succeeds, however, it's possible to advance to the next character of the regular expression and the next character of the text. This is done by calling `matchhere` recursively.

The situation is a bit more complicated because of some special cases, and of course the need to stop the recursion. The easiest case is that if the regular expression is at its end (`regexp[0] == '\0'`), then all previous tests have succeeded, and thus the regular expression matches the text.

If the regular expression is a character followed by a `*`, `matchstar` is called to see whether the closure matches. The function `matchstar(c, regexp, text)` tries to match repetitions of the text character `c`, beginning with zero repetitions and counting up, until it either finds a match of the rest of the text, or it fails and thus concludes that there is no match. This identifies a "shortest match", which is fine for simple pattern matching as in grep, where all that matters is to find a match as quickly as possible. A "longest match" is more intuitive and almost certain to be better for a text editor where the matched text will be replaced. Most modern regular expression libraries provide both alternatives, and TPOP presents a simple variant of `matchstar` for this case, shown later.

If the regular expression is a `$` at the end of the expression, then the text matches only if it too is at its end.

Otherwise, if we are not at the end of the text string (that is, `*text!='\0'`) and if the first character of the text string matches the first character of the regular expression, so far so good; we go on to test whether the next character of the regular expression matches the next character of the text by making a recursive call to `matchhere`. This recursive call is the heart of the algorithm and is why the code is so compact and clean.

If all of these attempts to match fail, then there can be no match at this point in the regular expression and the text, so `matchhere` returns 0.

This code really uses C pointers. At each stage of the recursion, if something matches, the recursive call that follows uses pointer arithmetic (e.g., `regexp+1` and `text+1`) so that the next function is called with the next character of the regular expression and of the text. The depth of recursion is no more than the length of the pattern, which in normal use is quite short, so there is no danger of running out of space.

Alternatives

This is a very elegant and well-written piece of code, but it's not perfect. What might one do differently? I might rearrange `matchhere` to deal with `$` before `*`. Although it makes no difference here, it feels a bit more natural, and a good rule is to do easy cases before hard ones.

In general, however, the order of tests is critical. For instance, in this test from `matchstar`:

```
    } while (*text != '\0' && (*text++ == c || c == '.'));
```

no matter what, we must advance over one more character of the text string, so the increment in `text++` must always be performed.

This code is careful about termination conditions. Generally, the success of a match is determined by whether the regular expression runs out at the same time as the text does. If they do run out together, that indicates a match; if one runs out before the other, there is no match. This is perhaps most obvious in a line like

```
        if (regexp[0] == '$' && regexp[1] == '\0')
            return *text == '\0';
```

but subtle termination conditions show up in other places as well.

The version of `matchstar` that implements leftmost *longest* matching begins by identifying a maximal sequence of occurrences of the input character `c`. Then it uses `matchhere` to try to extend the match to the rest of the pattern and the rest of the text. Each failure reduces the number of `c`'s by one and tries again, including the case of zero occurrences.

```
/* matchstar: leftmost longest search for c*regexp */
int matchstar(int c, char *regexp, char *text)
{
        char *t;

        for (t = text; *t != '\0' && (*t == c || c == '.'); t++)
                ;
```

Inner: 1798 x 984
Outer: 1798 x 1058

```
        do {    /* * matches zero or more */
                if (matchhere(regexp, t))
                        return 1;
        } while (t-- > text);
        return 0;
}
```

Consider the regular expression `(.*)`, which matches arbitrary text within parentheses. Given the target text

```
for (t = text; *t != '\0' && (*t == c || c == '.'); t++)
```

a longest match from the beginning will identify the entire parenthesized expression, while a shortest match will stop at the first right parenthesis. (Of course a longest match beginning from the second left parenthesis will extend to the end of the text.)

Building On It

The purpose of TPOP was to teach good programming. At the time the book was written, Rob and I were still at Bell Labs, so we did not have first-hand experience of how the book would be best used in a classroom. It has been gratifying to discover that some of the material does work well in classes. I have used this code since 2000 as a vehicle for teaching important points about programming.

First, it shows how recursion is useful and leads to clean code, in a new setting; it's not yet another version of Quicksort (or factorial!), nor is it some kind of tree walk.

It's also a good example for performance experiments. Its performance is not very different from the system versions of grep, which shows that the recursive technique is not too costly and that it's not worth trying to tune the code.

On the other hand, it is also a fine illustration of the importance of a good algorithm. If a pattern includes several `.*`s, the straightforward implementation requires a lot of backtracking, and in some cases will run very slowly indeed. (The standard Unix grep has the same properties.) For example, the command

```
grep 'a.*a.*a.*a.a'
```

takes about 20 seconds to process a 4 MB text file on a typical machine. An implementation based on converting a non-deterministic finite automaton to a deterministic automaton, as in egrep, will have much better performance on hard cases -- the same pattern and the same input is processed in less than a tenth of a second, and in general, the running time is independent of the pattern.

Extensions to the regular expression class can form the basis of a variety of assignments. For example:

(1) Add other metacharacters, like + for one or more occurrences of the previous character, or `?` for zero or one matches. Add some way to quote metacharacters, like `\$` to stand for a literal occurrence of `$`.

(2) Separate regular expression processing into a "compilation" phase and an "execution" phase. Compilation converts the regular expression into an internal form that makes the matching code simpler or such that subsequent matching runs faster. This separation is not necessary for the simple class of regular expressions in the original design, but it makes sense in grep-like applications where the class is richer and the same regular expression is used for a large number of input lines.

(3) Add character classes like `[abc]` and `[0-9]`, which in conventional grep notation match `a` or `b` or `c` and a digit respectively. This can be done in several ways, the most natural of which seems to be replacing the `char*`'s of the original code with a structure:

```
typedef struct RE {
        int     type;   /* CHAR, STAR, etc. */
```

```
            char    ch;     /* the character itself */
            char    *ccl;   /* for [...] instead */
            int     nccl;   /* true if class is negated [^...] */
    } RE;
```

and modifying the basic code to handle an array of these instead of an array of characters. It's not strictly necessary to separate compilation from execution for this situation, but it turns out to be a lot easier. Students who follow the advice to pre-compile into such a structure invariably do better than those who try to interpret some complicated pattern data structure on the fly.

Writing clear and unambiguous specifications for character classes is tough, and implementing them perfectly is worse, requiring a lot of tedious and uninstructive coding. I have simplified this assignment over time, and today most often ask for Perl-like shorthands such as `\d` for digit and `\D` for non-digit instead of the original bracketed ranges.

(4) Use an opaque type to hide the RE structure and all the implementation details. This is a good way to show object-oriented programming in C, which doesn't support much beyond this. In effect, one makes a regular expression class but with function names like `RE_new()` and `RE_match()` for the methods instead of the syntactic sugar of an object-oriented language.

(5) Modify the class of regular expressions to be like the wild cards in various shells: matches are implicitly anchored at both ends, `*` matches any number of characters, and `?` matches any single character. One can modify the algorithm or map the input into the existing algorithm.

(6) Convert the code to Java. The original code uses C pointers very well, and it's good practice to figure out the alternatives in a different language. Java versions use either `String.charAt` (indexing instead of pointers) or `String.substring` (closer to the pointer version). Neither seems as clear as the C code, and neither is as compact. Although performance isn't really part of this exercise, it is interesting to see that the Java implementation runs roughly six or seven times slower than the C versions.

(7) Write a wrapper class that converts from regular expressions of this class to Java's Pattern and Matcher classes, which separate the compilation and matching in a quite different way. This is a good example of the Adapter or Facade pattern, which puts a different face on an existing class or set of functions.

I've also used this code extensively to explore testing techniques. Regular expressions are rich enough that testing is far from trivial, but small enough that one can quickly write down a substantial collection of tests to be performed mechanically. For extensions like those just listed, I ask students to write a large number of tests in a compact language (yet another example of "notation") and use those tests on their own code; naturally I use their tests on other students' code as well.

Conclusions

I was amazed by how compact and elegant this code was when Rob Pike first wrote it -- it was much smaller and more powerful than I had thought possible. In hindsight, one can see a number of reasons why the code is so small.

First, the features are well chosen to be the most useful and to give the most insight into implementation, without any frills. For example, the implementation of the anchored patterns `^` and `$` requires only 3 or 4 lines, but shows how to handle special cases cleanly before handling the general cases uniformly. The closure operation `*` is a fundamental notion in regular expressions and provides the only way to handle patterns of unspecified lengths, so it has to be present. But it would add no insight to also provide + and `?` so those are left as exercises.

Second, recursion is a win. This fundamental programming technique almost always leads to smaller, cleaner and more elegant code than the equivalent written with explicit loops, and that is the case here. The idea of peeling off one matching character from the front of the regular expression and from the text, the

the rest, echoes the recursive structure of the traditional factorial or string length examples, but in a much more interesting and useful setting.

Rob has told me that the recursion was not so much an explicit design decision as a consequence of how he approached the problem: given a pattern and a text, write a function that looks for a match; that in turn needs a "matchhere" function; and so on.

> "I have pretty vivid memories of watching the code almost write itself this way. The only challenge was getting the edge conditions right to break the recursion. Put another way, the recursion is not only the implementation method, it's also a reflection of the thought process taken when writing the code, which is partly responsible for the code's simplicity. Most important, perhaps, I didn't have a design when I started, I just began to code and saw what developed. Suddenly, I was done."

Third, this code really uses the underlying language to good effect. Pointers can be mis-used, of course, but here they are used to create compact expressions that naturally express the extraction of individual characters and advancing to the next character. The same effect can be achieved by array indexing or substrings, but in this code, pointers do a better job, especially when coupled with C idioms for auto-increment and implicit conversion of truth values.

I don't know of another piece of code that does so much in so few lines, while providing such a rich source of insight and further ideas.

Acknowledgments

Your name here...