Regular expressions Part 1 of 2

Nicholas Cutler

The second article in my *Secret Life of Algorithms* series (Archive 22:9) looked at the problem of finding a word within a string of text, rather like the search-and-replace function in a wordprocessor. What's called "regular expressions" extends this idea into a general tool for processing text. Regular expressions are commonly supported by the more advanced text-editors aimed at programmers, as well as by standalone utilities and some newer programming languages.

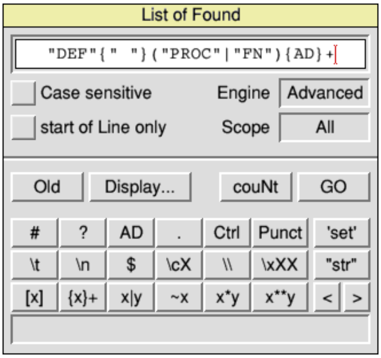
To give a more realistic example, I was once faced with the task of taking formatted text from Interword, a BBC Micro wordprocessor, and importing it into !Ovation on the Archimedes. Using a text-editor that supported regular expressions enabled me to solve the problem without writing a program.

Some baby steps

To begin with, suppose you are trying to find every occurrence of the word *walk* in a document and replace it with *run*. This is a simple enough task, but if you're not careful a naive search would also find references to *walkers crisps*! You can attempt to solve this problem by searching for every occurrence of *walk* followed by a space, but regular expressions allow us to be more specific and consider some of the other characters – such as newline or tab or close-bracket or close-quote or emdash or comma or fullstop – that may mark the end of a word.

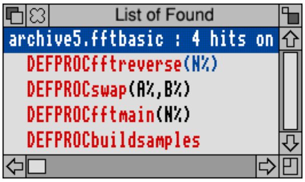
Like wildcards

Many readers may already have some experience of simple regular expressions, as the idea is commonly used for specifying patterns in text. Filecore allows the use of certain "wildcards" when specifying filenames to commands like `copy` and `count`. An asterisk means any sequence of characters, so `count */jpg` will count the total size of all files with the suffix "/jpg". If they were files on a Dos-type filing system this would simply mean all Jpeg images. Similarly, a hash stands for any one character, so `count code/#` would match all files called "code" that have a single-letter suffix, usually things like C or assembler source code (again if you are following Dos conventions). Once again, regular expressions let us specify much more complex patterns if necessary.


List of Found

```
"DEF"{" "}("PROC"|"FN"){AD}+
```

Case sensitive [ ] Engine: Advanced
start of Line only [ ] Scope: All


List of Found

```
archive5.fftbasic : 4 hits on
 DEFPROCfftreverse(N%)
 DEFPROCswap(A%,B%)
 DEFPROCfftmain(N%)
 DEFPROCbuildsamples
```

StrongEd example

The idea of regular expressions is still more common on Linux than on RISC OS, but we do have a number of tools that support them. The search option with wildcards in !Edit is an example; other text-editors like !StrongEd go further.

To give you an idea of what's possible, start by loading a Basic program into !StrongEd. Press F2 to get the "list of found" dialogue, ensure that you've selected the advanced option and enter the following into the search:

```
"DEF"{" "}("PROC"|"FN"){AD}+
```

[Note the distinction between ordinary and curly brackets. —Ed.] This specifies a pattern that searches the text of the program for all function or procedure definitions. If you now run this search, another window will open listing all such lines in your program. You should see something like the first screenshot.

A general module, from Perl

This is a useful enough feature in its own right, but part of the power of regular expressions comes from the ability to use them in your own programs to automate text-processing tasks. For this purpose, the Regex module can be used from most programming languages, while languages such as Perl offer them as a core feature of the language. The exact set of regular expression features offered by these tools varies, as does the syntax, but once you've learnt one, you'll find the idea is similar for others.

Throughout this article I will focus on the regular expressions used by Perl, principally for the range of facilities which they offer, but also because it is the one with which I am most familiar. Perl is commonly used on Linux for automating a lot of text-processing tasks, and there is also a RISC OS port.

More conveniently there is also the PCRE (Perl-Compatible Regular Expression) library, which provides all of Perl's regular expression facilities, and which can be used from within programs written in C. To support this article I have ported this library and provided a simple command-line tool called Regex to search a textfile for a specified pattern. This will let you try out regular expressions as you read through this article.

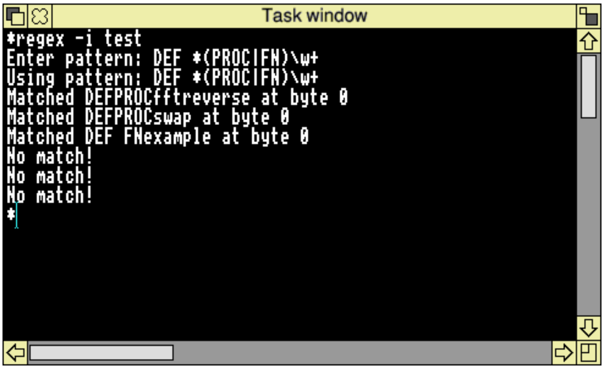
To use this program you will need to prepare a textfile with a number of lines, at least some of which should match your pattern. To try this out, change to the directory containing Regex and type

```
*regex -i test
```

When prompted for the pattern, enter:

```
DEF *(PROC|FN)\w+
```

Like the example in StrongEd, this searches for procedure or function definitions in Basic, but this time using Perl's regular-expression syntax. If you try this rather contrived example in a task window you will see something like the screenshot here.



```
*regex -i test
Enter pattern: DEF *(PROC|FN)\w+
Using pattern: DEF *(PROC|FN)\w+
Matched DEFPROCfftreverse at byte 0
Matched DEFPROCswap at byte 0
Matched DEF FNexample at byte 0
No match!
No match!
No match!
*
```

Throughout this article you will find other examples to try. You can download these and the Regex program from the Archive website.

Files: see p.2

The basics

To begin with, recall that a regular expression is just a way of specifying a pattern that you want to find within some text. At a basic level most characters, including all the letters and digits, have the expected meaning. One letter on its own will find every occurrence of that character wherever it may appear. A sequence of letters will find every location where that string occurs, whether on its own or as part of another word. In this way, simple expressions work just like the most basic search facility in a wordprocessor, or the `INSTR` function in BBC Basic.

Metacharacters

To extend the concept, regular expressions give some symbols a special meaning, and these are known as "metacharacters". For instance, a dot "`.`" does not match a full-stop, but rather any single character. In this way it is analagous to using the hash # in filing-system commands.

Similarly, an asterisk means match any number of the previous character, so

```
a*
```

will match any number of the letter "a" in a row, and

```
.*
```

matches any sequence of characters. Note that this is not quite the same as the usage in filing-system commands I mentioned above.

Finally, a backslash "\" is used as an *escape* character to modify the behaviour of the following letter or symbol. So if you need to match a literal full-stop, use

```
\.
```

There are a few other metacharacters and also other situations where the escape character is used. Read on for examples.

Character classes

If you have tried some simple expressions then you will be aware of the problem of constraining the range of possible matches to include only the ones you actually want. To return to the earlier example of matching a single word, there is nothing to stop the computer finding an occurrence of the word as part of another. To improve this we can try to find only those locations where the desired word is surrounded by spaces. But this excludes any locations where the word comes at the end of a sentence or is followed by a punctuation mark. To correct this we need to be able to say something like "match any one member of a set of characters".

To solve just this type of problem, regular expressions introduce the idea of character classes. It's rather like the instruction to match any character, but more specific. Classes enable us to do things like find any single digit. Thus, for example, `\d` matches any digit, but does not match a letter or punctuation mark.

Other commonly supported character classes include `\s` for any whitespace character, including tabs and linefeeds; `\w` matches any "word character" (basically letters and digits).

Changing the lowercase letter to

Regex on Ro

If you want to use regular expressions from within your own programs, two choices are available on RISC OS:

◆ The Regex module (*sbellon.de/sw-modules.html*) is based on the Gnu regular expression library, but it is wrapped up into a RISC OS module and invoked using SWI calls. It was first written by Neil Bird; Stefan Bellon took up maintenance from version 1.03. Although it is a little old now (the latest is 1.06 in 2006) it has the advantage that it can be used from within almost any programming language.

◆ Alternatively, I have ported Philip Hazel's PCRE, the Perl-Compatible Regular Expression library (*pcre.org*). This is much more recent and offers more facilities, including Unicode support, but at present can only be linked with C programs. I hope to publish it with Part 2 of this article.

uppercase gives the complement of a particular class. So `\D` then matches anything which is *not* a digit; `\S` and `\W` work in a similar way. It is now possible to write a regular expression to look for a single word, and reliably exclude those cases where the word is part of another. So

```
\Wwalk\W
```

finds any occurrence of the word walk surrounded by "non-word" characters, including spaces and punctuation symbols.

Make up your own classes

This expression is probably good enough in this example, but if we wanted to be more specific it is possible to make up new classes. To do this simply enclose the range of permitted letters or symbols in square brackets. Thus, `[abc]` means any one of the letters *a*, *b* or *c*.

Likewise, returning to the single-word example, if we knew that words would be followed only by a space, full-stop or comma, then `[ .,]` would be sufficient in place of the `\W` expression. Note that the full-stop does not need to be escaped in this case, because inside a class it reverts to its usual meaning.

Another feature is the ability to specify a range of characters, making it more straightforward to build quite large classes. Thus `[0-9]` means any character between 0 and 9 inclusive – this is a more explicit equivalent to `\d` in an expression.

Finally, to define the complement of a class, add the "^" symbol just inside the opening bracket: `[^0-9]` means anything *except* the digits 0 to 9.

A practical example: finding hex

With some variations in notation, character classes are found in almost all regular-expression systems and, unlike the simple wildcards allowed in filenames, they enable us to be quite specific about what letters are acceptable at a given point. With this in mind it is now possible to look at a practical example:

Suppose you wish to find all of the hexadecimal numbers in the text of a program. This is a fairly typical use of regular expressions and is enough to justify their inclusion in many text-editors aimed at programmers.

First, to match a single hexadecimal digit, we can use this character class:

```
[0-9A-F]
```

Recall that the asterisk symbol means "match any number of the previous digit". Adding this to the class gives the expression

```
[0-9A-F]*
```

which really means "match a sequence consisting of any number of the digits 0-9 or A-F".

This is a good start but it is not sufficient on its own, because it will quite happily match `FAFF`, either on its own or as part of another word. To make the expression more specific we need to add additional text that will serve to delimit a hex number. In this case, we note that, at least in the C programming language, hexadecimal numbers always begin with "0x". Adding this to the begining of the expression gives

```
0x[0-9A-F]*
```

which then accepts genuine hex numbers, but rules out any words consisting solely of the letters A to F. For most practical purposes this is a reasonable attempt.

If you wish to try this example, you can, again, use Regex. This time the pattern is included in the data file so you need type only `*regex hexnum`.

Finding IP addresses

Consider another example, namely that of matching numeric internet addresses. These are normally written as four numbers between 0 and 255 separated by full-stops. We can easily match a single digit using `\d` and a dot with `\.` – so adding an asterisk after each `\d` seems a reasonable guess. This would certainly match all valid IP addresses, but, unfortunately, it is not specific enough. Recall that the asterisk simply means "any number of". That includes "none at all" and "more than three". Thus a simple attempt would also find numbers that are far too long and would even find three dots in a row!

To improve matters there are often additional metacharacters that work a bit like the asterisk but allow us more control. In Perl regular expressions, we can use a plus sign "+" to mean "one or more of the previous character", and a question-mark "?" for "either zero or one of the preceeding symbol". To put this another way, the plus works just like the asterisk but insists that at least one copy be present; the question-mark makes a character optional.

To return to the example of IP addresses, replacing the asterisk with a plus will, at least, reject the case of three consecutive dots. However, the problem of matching numbers that are too long remains. To eliminate this, there is one further quantifier, consisting of two numbers in braces, which will allow us to be arbitrarily specific. In this particular case, we can write {1,3} to mean between one and three, easily matching the valid numbers in an IP address. So the complete expression is now:

```
\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}
```

Alternates

The above has introduced character classes and quantifiers. Used together these allow us to specify some useful patterns for matching text, as the two examples demonstrate. This has shown how to match a specific word, an arbitrary word using something like

```
\w+
```

or any number of a specific letter.

However, suppose we needed to match any one of a number of words from a list. Regular expressions also let us specify alternates, rather like the OR condition in many programming languages: "either this or that". As we have already seen in the example of PROC or FN names, this is often written using the vertical-bar character "|". Thus the expression

```
walk|run
```

matches either of the two human gaits. If we only want to match one or the other on its own, and not as part of another word, then as before, we can add `\W` to insist that the match begin and end with a space or punctuation mark. In this case, however, it becomes necessary to add brackets round the alternate words to make it clear that `\W` is not part of the alternation, and should always be present at the begining and end. Thus, the complete expression

```
\W(walk|run)\W
```

should suffice.

Finding domain names

It is now possible to look at another realistic example: matching internet domain names. Consider something like *archivemag.co.uk* – it has three parts, the name of the organization, its type (like *.co* for company), and the country.

The first section is easy enough to match: an arbitrary number of word characters should work. For the second there are only a limited number of options, so this is a perfect use of the alternation facility. The final part, the country code, presents a bit more of a problem: a very long list of alternatives is possible, but would be tedious, and presents problems if a new country were created. In this case it is probably sufficient to accept any two lowercase letters. So, a complete expression would look something like:

```
\w+\.(co|ac|org|gov)\.[a-z]{2}
```

This isn't perfect, of course: some domains, particularly American ones, don't have a country code on the end, and still others omit the type of organization.

An additional exercise for the reader is to extend this expression to catch the other possibilities too. As a hint, try using another alternation with the whole of the above expression as one of the possibilities.

Mind the brackets

Notice that the ordinary round brackets fulfil a dual role. As seen above, they can limit the scope of the alternation, rather like brackets in a mathematical expression designate sections that must be evaluated first. In a similar way, they can be used with a quantifier to match whole sections of a pattern that may be repeated, rather than just single characters or classes.

The other meaning of the brackets has so far been hidden. This other meaning is to "capture" which part of the string matched the expression. In the domain-names example above, it would tell us which of the four alternatives would be found. When this expression matches against *archivemag.co.uk*, the brackets match the "*co*" section. This enables us to refer back to parts of the matched string, which is especially useful when replacing sections of text, or for performing further processing in a program.

To illustrate this, try this example again with the command line

```
*regex -c Domains
```

As another example consider

```
color=#([0-9a-f]+)
```

which may be used to match the colour property within an HTML tag. In this case the round brackets capture the hexadecimal colour number for further use, possibly as part of a program that processes HTML – you could produce a list of all the colours you use in your website, for instance. Or you could refer back to the captured value within the regular expression itself. I hope to cover this "back-reference" facility in Part 2.

As a final example, consider the problem of a section of text that is entirely optional. If this were only one character, we can use a class with a quantifier, like

```
[+-]?
```

But suppose numbers were instead written optionally with either of the words *plus* or *minus*. Here brackets will enable us to combine alternation with a quantifier, as in

```
(plus|minus)?\d+
```

Warning

A warning to end with: it is possible to nest quantifiers. While this is doubtless a powerful facility, it makes it possible to specify expressions that can take a very long time to match.

Summary and next

This article has introduced the concept of regular expressions, and given a summary of the most commonly used features, together with examples. If your main use is to match patterns in text, then this should suffice to get you started.

Beyond this, however, a planned sequel will look at some more advanced features, as well as questions of efficiency, and using regular expressions in your own programs.

Nicholas Cutler ncc25@cam.ac.uk

Nicholas recently started bellringing again when he heard the bells being rung in his local parish church. Evidently, there are also many other campanologists among RISC OS users.