Processing Text Using Regular Expressions

How New System Operators to Search and Replace Text Were Devised

Richard Smith

Dyalog Ltd

Minchens Court, Minchens Lane, Bramley, RG26 5BH, United Kingdom

richard@dyalog.com

Main topic: State of the art of array processing languages

Abstract

A fundamental component of scripting languages (Perl and Tcl, for example) is the ability to search text using **regular expressions** - a means of describing complex patterns of characters within text. Although APL is traditionally used to process numeric data it also has the characteristics of a good scripting language, and current and future APL applications could benefit from the power and flexibility offered by support for regular expressions which is tightly integrated into the language.

This paper discusses the design decisions which led ultimately to the system operators which integrate regular expressions into Dyalog APL. With these, APL users can search text and make modifications, using either a simple expression or a powerful APL function to express the transformation.

Overview

A regular expression (also called a 'pattern') describes a set of character strings, and one or more regular expressions can be used to locate strings within a text document[1]. For example, the simple regular expression '.at' matches all sequences of three characters ending with the letters 'at' and this pattern matches all three of the underlined parts within 'The cat sat on the mat'. The dot is an example of the regular expression syntax which describes a class of characters (here, *any* character). Character classes can be modified to specify repetition, alternates and other such properties. The power of regular expressions is the ability to describe complex sets concisely and simply.

Once the strings within the document have been identified there are potentially many different things you may want to do with them, which fall into one of two categories: **replace** and **search**. Some examples are:

Replace

- Modify a document by transforming the strings in some way

[1] This paper distinguishes the **entire searched text** and the sections of **text which matches the regular expressions** by referring to them as the **document** and **strings** respectively.

Search

- Identify the positions of strings within a document
- Generate a vector of the strings themselves
- Generate a vector of the strings themselves, each transformed in some way
- Extract from the document entire lines in which the strings were found, and discard the others (or the reverse)

Existing scripting languages and editors derive great power from being able to perform one or more of these tasks. Examples include:

Bash

The command syntax used in the Bash shell allows comparisons to include regular expressions. For example, to test whether the value of variable 'a' contains a sequence of any three consecutive digits anywhere within it:

```
if [[ $a =~ [0-9]{3} ]] ; then
        echo yes
fi
```

'[0-9]' matches any character between 0 and 9
'{3}' repeats this three times

Perl

Perl allows searches (matches) much like Bash, above, plus replaces (substitutions and translations). To parenthesise the first letter in every word within a variable you could write:

```
$string =~ s/(\w)(\w*)/($1)$2/g;
```

'/' separates the search pattern, transformation and options
'\w' matches any "word" character
'*' repeats this zero or more times
'(' and ')' form groups within the sequence, and '$1' and '$2' in the transformation expand to whatever the corresponding group matched.

grep

grep is a command which filters lines from a file, retaining only those which match the given regular expression. For example, to extract only lines which have the character '#' as the first non-whitespace character on them:

```
grep "^\s*#" myfile
```

'^' anchors the match to the start of the line
'\s' matches any "space" character
'*' repeats this zero or more times

vi editor

The vi text editor uses regular expressions for both search and replace. The following replace command includes word delimiters within the pattern so that it replaces the word ‘len’ with ‘length’, but not words which contain ‘len’ within them:

```
:%s/\<len\>/length/g
```

‘/’ separates the search pattern, transformation and options
‘\< ’ and ‘\>’ delimit the match with anything (including start/end of the line) which is a ‘non-word’ character

With the following C code it changes the three occurrences of the variable ‘len’ but *not* the variable ‘save_len’:

```
save_len = len;
len = fn (len);
```

Not only would all APL users benefit from similar functionality within the language, it is now so prevalent that *not* having it would be seen as an obstacle to encouraging new users to take up APL.

The PCRE (Perl Compatible Regular Expressions) library[2], which is used in many open source and commercial projects, provides a set of functions that can be used by the interpreter to implement regular expression pattern matching. It performs the bulk of the work needed for a search or replace operation and the decision to use it rather than implement such a library from scratch was an easy one. There are some potential downsides to using PCRE – these are noted in the following sections of this document – but they are not considered strong reasons to avoid using it.

Key requirements

The tools and languages cited in the previous section can each perform a different range of actions using regular expressions but are generally tailored to perform a particular task. A primary requirement for APL was to devise something which was general-purpose enough to meet all the different search / replace requirements and major decisions had to be taken on how this functionality should be presented to the user. However, all cases are essentially comprised of the following common elements:

- The document to be searched
- One or more regular expressions to identify strings within the text
- Transformation rules to determine how to process the strings

Representation of documents

Regular expressions operate over documents – one or more lines of text. One fundamental design decision which had to be made was how a document should be represented within APL. It

[2] http://www.pcre.org/

became clear that a text document would almost exclusively be imported or created in the workspace in one of two main ways:

1. As a single character vector, with embedded line-ending characters within the text, and

2. As a vector of character vectors, with each character vector representing a single line of text.

These are the two fundamental array types which it was decided to process. These are different ways of representing the same thing, so it was considered important that the selected format should be preserved - that is, when replacing text, the transformations may add or remove lines, but the format of the input should remain the same on output. It was decided to “tolerate” a combination of the two - a vector of character vectors which also contains embedded line ending characters - and in this case the input is implicitly “normalised” as if it had been presented as a vector of character vectors. Character matrices were considered as a third possible format, with each row representing one line, but these were ultimately rejected because of the complications caused by spaces at the ends of lines.

It was also considered useful to be able to process documents which are not already in the workspace. The obvious example is a file, but this idea can be extended to any stream of data. The advantage of processing streams is that it introduces the possibility of filtering large quantities of data without having to bring it all into the workspace first. At the time of writing support for streams has not been finalised; as an interim measure data can be read from and written to tied native files. A complication in processing data from external sources is that the data encoding has to be known or specified, and handled appropriately.

Processing documents in their entirety or line-by-line

The PCRE search engine processes blocks of text and is used to create a list of matching strings within that text. It is advantageous to be able to process documents in two different ways: line-by-line (the document is split into separate lines and each line is processed separately) or in their entirety (the entire document is passed to the engine in one go). The advantages of the two methods are summarised as follows:

Line-by-line

- Less space is required in the workspace – particularly when processing documents from external sources; arbitrarily long streams of data may be too big to store in their entirety
- Line-based searches are often useful – for example, one may wish to locate only the first match on each and every line (not every match, and not just the first in the entire document)

In their entirety

- Regular expressions may operate over more than one line
- Replace operations may completely eliminate lines from the document (rather than just reduce the line to zero length)

One aspect of the design which was difficult to resolve was the distinction between how a document is *processed* (in its entirety or line-by-line) and how it is *specified* (as a single character vector or a vector of character vectors). There were two schools of thought:

- The input format dictates the processing format: a single character vector (whether it contains line-ending characters or not) is processed in its entirety and a vector of character vectors is processed line-by-line; external stream data is selectable.
- The input format and the processing format are unrelated. All input formats, including streams, are just different representations of the same thing, and the processing of the document should not be dictated by the representation.

The latter view has been considered counter-intuitive by some, but it does offer the greatest flexibility. It was therefore decided that the processing format and input format would be independent, and an option would be provided to select whether the document should be processed line-by-line or not.

Note that PCRE itself has a line mode, which is not the same thing.

Specifying transformations

Transformations specify how the matching strings within the document should be modified. It is clear some transformation is needed for a replace operation - without it, the document would remain unchanged. A transformation can still be useful for a search operation, and can be far more radical - it is not limited to generating character strings which are substituted back into the original document - and could generate the positions of the matches rather than the matches themselves, counts of some kind, or indeed any values at all.

Traditional tools often use a syntax similar to the regular expression to specify the transformation. For example:

- A replace with the regular expression ‘red’ and the transformation ‘blue’ would change all occurrences of ‘red’ to ‘blue’
- A search with the regular expression‘(.)at’ and the transformation ‘\1’ would find all sequences of three characters ending with ‘at’ and return just the first character. For ‘The cat sat on the mat’ it would find ‘c’, ‘s’ and ‘m’.

Clearly, this type of transformation syntax is needed. However, it may be desirable to perform far more complex transformations than this could offer - for example, to convert monetary values identified within a text document to a different currency using a mathematical formula. A fundamental concept in the design of Dyalog’s regular expression support, therefore, was that it should be implemented as an operator so that a function may be used to perform the transformation using the full power of APL code.

Implementing an operator rather than a function had a very significant impact on the implementation because the search no longer executes as a single atomic function. That is, calling APL during the search allows for searches to be nested within searches, the workspace to be saved and reloaded, and APL to switch threads, amongst other things – all of which complicate the way the PCRE engine can be used.

In order to perform a transformation the function would need to be given a large quantity of information about the match – for example, the matching text itself, the groups within it, and its position within the document. It was considered too cumbersome to simply bundle these up into a vector, so the adopted approach was to name each value and pass a namespace containing these values. At present this namespace contains ten different items, eg `Match` for the matching text, `Pattern` for the pattern which resulted in the match and `Groups` for a vector of the names of the groups within the pattern. A simple transformation which reverses all matching text could therefore be implemented as:

```
{⌽⍵.Match}
```

A function makes it possible to construct just about any transformation but does not displace the simple syntax because that remains the easiest way to construct simple textual transformation. In fact, another relatively simple and commonly performed search task is to simply locate matching text within a document by position and length and to simplify this a third transformation type was provided which uses numeric codes to indicate numeric result types: 1 for line position, 2 for length, etc.

Search optimisation

Searching text using regular expressions is complex and potentially quite slow. Whilst the performance of the search engine itself is beyond our control, there are some things that can be done to help.

PCRE requires that a regular expression be presented to it for compilation before it is used for searches. This process takes time. It can also optionally undergo a “study” (optimisation) phase which takes further time, but which may then improve the search performance. A simplistic implementation would perform each search by (a) compiling the regular expression, (b) optimising the regular expression (or avoiding this step altogether), (c) proceeding with the search, and (d) discarding the compiled regular expression when the search was complete. If the search was in a loop (meaning that the same regular expression would be used over and over again) it would make sense to avoid repeating the compile/optimise step each time. We can improve this simplistic implementation by retaining the compiled regular expressions between searches – even holding just the last compiled regular expression and discarding it only when a different one is encountered was found to be worthwhile. Of course, deciding when to discard and when to retain the compiled form does itself have a slight performance impact and the simplistic implementation wins in some circumstances; the best balance was determined to be obtained from caching a relatively small number of compiled patterns.

Deciding when and when not to optimise the regular expression is more difficult. The interpreter does not know how “complicated” the search would be and whether there would be any overall benefit. It could make some assumptions based on the length of the expression and the document to be searched, but the method chosen was this:

- On first encountering a regular expression, just compile it.
- When a regular expression is encountered a second time (i.e. is found in the cache) assume it will be used many times and optimise it.

Syntax

Having determined the functionality required of the new operators, several factors had to be considered when devising suitable syntax.

Primitive or system operator

The operator is an interface to an external tool - and one which operates only on specific array types rather than any data in general. Because of this it became clear that this functionality should not be implemented as a primitive but should instead be a system operator. This has the advantage that a new glyph is not needed, although § had been considered for the purpose (representing “string search” – terminology which has been retained). Up until this point, Dyalog had no system operators (just system functions and system variables) so this decision represented a bit of a milestone.

Operator name

Choosing a suitable name for the new operator was one of the most contentious design decisions of all. Search, and search/replace are closely related, and initially it was intended to have a single operator which performed both (with an option to specify which). `⎕SS` is used in other APL dialects for string search and replace, but they are fundamentally different implementations and this name was considered inappropriate. `⎕PCRE` and `⎕REGEX` were considered, but finally `⎕RX` (for Regular eXpression) was chosen. Once implemented, however, it became apparent that two separate operators – one for search, and one for replace – would be far more convenient. All sorts of problems were encountered in choosing new names: they had to be descriptive but not too long and some obvious name candidates (such as `⎕SR`) are already in use. There were arguments over the exact meanings of terms such as “search” and “replace”, and every synonym thereof. In the end, `⎕R` and `⎕S` were chosen.

Argument and operand order, and problematic options

`⎕R` and `⎕S` require:

- One or more regular expressions
- Options, which in general affect the search so primarily associate with the regular expressions
- Transformation rules, specified as character or numeric vectors, or a function
- The document to process, or a reference to a stream or file
- Optionally, a reference to a stream or file to send the output to.

Distributing these amongst the operands and arguments is mostly straightforward, but options were problematic. The operands need to include both the regular expressions and the transformation rules so that the derived function can be named and used to transform data. It would be syntactically invalid to create a single operand which contained character vectors and a function, so these together occupy both right and left operands. Then, the arguments to the derived function are the input document (right) and optionally the output stream (left):

```
optional-output ( regular-expressions ⎕R transformations ) input
```

Options *generally* affect the search – they specify whether searches are case sensitive, for example, or whether documents are processed in their entirety or line-by-line. Therefore they most logically belong with the regular expressions, and this is where initial implementations had them. But there are some problems with this: it is cumbersome to specify them in this way and, more importantly, not *all* options affect the search – some concern the document, such as file encoding. It is not ideal that a derived function which transforms data in some way also carries assumptions about the data it will be used with.

When discussing the APL# language for the paper "APL# - An APL for Microsoft.Net, Mono, SilverLight and MoonLight", Roger Hui reminded us of the variant operator (⍠[3]) which Ken Iverson had proposed as a means of selecting variants of primitive functions (for example, generating indices with different index origins), and this is also ideal for this purpose.

The variant operator takes a function as its left operand and an array of option values as its right operand. This gives the opportunity to specify options independently, for example you may have a derived function which searches for the character immediately preceding the letters 'at':

```
f←'(.)at' ⎕S '\1'
```

To set an option to make searches case insensitive this function can be used with an option:

```
(f ⍠ 'ignore-case') 'The CAT sat on the mat'
```

or the function itself could carry the option:

```
fi←'(.)at' ⎕S '\1' ⍠ 'ignore-case'
```

or a combination could be used:

```
(fi ⍠ 'file-encoding' 'utf-8') tienum
```

In this last example, both ignore-case and the file-encoding options are specified.

Dyalog has several existing functions which take options – ⎕FCHK and ⎕XML for example – which currently have no consistent syntax. ⍠ provides the opportunity to create a more uniform and more flexible option syntax, and it is anticipated that future releases will make further use of it – both for new functionality, and as an alternative for that which already exists.

Examples of use

The following simple examples illustrate the basic functionality of the ⎕R and ⎕S operators:

[3] The symbol selected for variant will probably be ⍠, but as an interim measure (mostly because the classic variant of Dyalog APL does not have space for the adoption of new symbols) there is a synonymous system operator, provisionally named ⎕OPT.

Replace operations

1. Using a transformation pattern:

```
      ('.at' ⎕R '\u0') 'The cat sat on the mat'
The CAT SAT on the MAT
```

   In the search pattern the period matches any character so the pattern as a whole matches sequences of three characters ending ‘at’. The transformation is given as a character string, and causes the entire matching text to be folded to upper case.

2. Using a transformation function:

```
      ('\w+' ⎕R {⌽⍵.Match}) 'The cat sat on the mat'
ehT tac tas no eht tam
```

   The search pattern matches each word. The transformation is given as a function, which receives a namespace containing various variables describing the match, and it returns the match in reverse, which in turn replaces the matched text.

Search operations

3. Using a transformation pattern:

```
      ('.at' ⎕S '\u0') 'The cat sat on the mat'
 CAT  SAT  MAT
```

   The example is identical to the first, above, except that after the transformation is applied to the matches the results are returned in a vector, not substituted into the source text.

4. Using a transformation function:

```
      ('.at' ⎕S {⍵.((1↑Offsets),1↑Lengths)}) 'The cat sat↲
on the mat'
 4 3  8 3  19 3
```

   When searching, the result vector need not contain only text and in this example the function returns the numeric position and length of the match given to it; the resultant vector contains these values for each of the three matches.

5. Using numeric transformation codes:

```
      ('.at' ⎕S 1 2) 'The cat sat on the mat'
 4 3  8 3  19 3
```

   Here the transformation is given as a vector of numeric codes which are a short-hand for the position and length of each match; the overall result is therefore identical to the previous example.

Future developments

The use of an off-the-shelf search engine has many benefits aside from being immediately ready to use and license-free. One major advantage is that the search pattern syntax is near-universal and the library is rich with features; the flipside to that, however, is that the pattern syntax and the search functionality on offer are predetermined and inflexible. One particular area Dyalog is researching is the provision of additional character classes designed to enable the processing of APL code. It would be useful to be able to search through the workspace or particular functions in order to locate particular named items or rename certain arrays, for example. Simplistically, one could use an expression such as:

```
⎕FX ('END' ⎕R 'XYZ') ⎕CR 'f'
```

This modifies the function `f` so that all occurrences of the identifier `END` are replaced with `XYZ`, but this is insufficient - in this form it will also incorrectly modify:

```
'THE END IS NIGH'
:END
TENDER
∆END
```

PCRE already supports a rich set of character classes (such as `\w`, which matches any "word" character) but APL clearly has a larger character set than most other languages and it is likely that APL source will provide a challenge. For example, a pattern of the form `\w+` will match "words" (sequences of one or more of the characters `a` to `z`, `A` to `Z`, `0` to `9` and `_`), which is ideal for searches through languages such as C, but an APL identifier is not limited to this set of characters. `\w` is equivalent to explicitly constructing the class, for example:

```
[a-zA-Z0-9_]
```

An equivalent for Dyalog APL needs to include delta and accented and other European characters, so in its complete form is:

```
[_a-z∆A-Z⍙ÁÂÃÇÈÊËÌÍÎÏÐÒÓÔÕÙÚÛÝþãìðòõÀÄÅÆÉÑÖØÜßàáâäåæçèéêëíîïñóôöøùúûü0-9]
```

Whilst a search pattern could be constructed using classes in this form, it would be cumbersome and error-prone; it would be far more convenient to have a predefined character class - say `\⍳` - as a short form. This could be implemented by modifying the engine itself (the sources are available) or alternatively by modifying user-provided search patterns within the interpreter before they are processed by the engine.

This character class is still insufficient for isolation of identifiers within APL source. For example:

```
('\⍳+' ⎕R {⍵.Match≡'END':'XYZ' ⋄ ⍵.Match})
```

would modify `END` and correctly *not* modify `TENDER` or `∆END`, but it would still erroneously modify

```
'THE END IS NIGH'
:END
```

The pattern `'\⍳+'` also matches sequences beginning with, or consisting entirely of, numeric characters - which means it will improperly match the identifier `Z` in this code[4]:

```
A B C ← 1Z 3
```

Resolving these problems is possible but it involves quite a complex search pattern containing separate character classes for leading and subsequent characters, look-behind assertions and non-capturing subgroups. Again it seems that an APL-specific class would seem useful and quite possible to implement, although a class which matches a sequence of characters rather than a single character would be without precedent and may have unexpected consequences.

Dyalog is still researching this area, and would welcome involvement from the APL community.

[4] Whilst this is legal APL code, `⎕CR` will reconstruct this with a space between the `1` and `Z`, conveniently avoiding the problem.