Regular expressions Part 2

Nicholas Cutler

The previous part of this series (*Archive 23:10*) introduced the concept of regular expressions and looked at the core features that they provide for matching patterns in text. This part will take the concept further by looking at some more advanced features and will conclude by showing how you can use the regular expression library in your own programs.

Unicode subsets

So far, we have assumed that a letter in the regular expression, or in the string we are searching, is just one entity from the computer's character set, represented by a single byte. For most of the time this is a reasonable assumption, but if you ever have to handle languages other than English, or handle non-Roman scripts, then you will probably have come across Unicode. Put simply, Unicode aims to include every character you can think of, from the familiar Roman alphabet, through Greek and Cyrillic, to all of the Chinese and Japanese ideographs. Although support for Unicode in $R_{ISC\ OS}$ is rather patchy, PCRE regular expressions do support it. *[See box below]*

Not surprisingly, with so many extra letters and symbols to cope with, the idea of a character class has grown somewhat. To begin with, recall that `\s` would match any space character, including things like tabs. This is generally fine, but Unicode adds a whole host of extra spacing characters. If we want to include all of these too then we can use the expression `\p{Zs}`. In this case, the `\p` means "any Unicode character with a specific property", and the parameter in braces is the code for spacing characters. There are many other Unicode properties, like `\p{Ll}` for lowercase letters and `\p{Sc}` for currency symbols. For a full list, refer to the PCRE documentation.

Hazel grew from Acorn roots

PCRE stands for the Perl-Compatible Regular Expressions library. It was written by Philip Hazel of the University of Cambridge Computing Service, starting in 1997. PCRE has an exceptionally powerful and flexible syntax, and the library is often incorporated into open-source programs such as the Apache HTTP server and PHP scripting language. *pcre.org*

Nicholas Cutler has now ported PCRE 10.10 to $R_{ISC\ OS}$ as a relocatable module that can be called via SWIs from almost any language. *www.cl.cam.ac.uk/~ncc25/pcre_riscos.zip*

Philip Hazel grew up in the Acorn world, where he wrote a music-typesetting program called **PMS** (Philip's Music Scribe; a commercial version of it was called Scorewriter). It is written in BCPL, a very rare language for an Archimedes program, because its initial platform was Acorn's Cambridge Workstation, which predates the Archimedes computer. Richard Hallas reviewed PMS in *Archive 5:11*. Philip continues to develop it on Unix-like platforms, where it is called **PMW** (Philip's Music Writer).

When he retired from the university, Philip wrote *From Punched Cards to Flat Screens*, where his life story just about coincides with that of computers. The memoirs are available as PDF from *people.ds.cam.ac.uk/ph10* – the music program is there too.

If we combine this with some of the facilities explained in the previous article, we can look at another practical example: matching monetary values. Firstly, these should begin with a currency symbol, followed by one or more digits. Next, there may be a decimal point and two further digits. This gives the expression:

```
\p{Sc}\d+(\.\d\d)?
```

In addition to matching characters with a specific property, the `\p` notation can also be used to match any character within a given script or alphabet. In this latter case, there are no codes to remember, and PCRE will recognize the usual name for the script. So, for example, `\p{Cyrillic}` can be used to match any letter in the Cyrillic alphabet. The names used for most scripts should be familiar, but, once again, if you need a full list then consult the PCRE documentation.

At this point it is worth mentioning that one particular script does not always contain all characters used by that language. Very often, common spacing and punctuation symbols used in several languages or scripts are located elsewhere. In the above example, for instance, all of the necessary punctuation marks are instead found in the familiar Latin character set. If you need to match these "shared" characters then you can use the notation `\p{Common}`.

Negating a property

In common with the idea of negating a character class, a similar principle applies with character properties. So just as `\s` means any space character, and `\S` is the opposite (anything which isn't white space), `\P` matches anything without the specified property. Thus, for example, `\P{Ll}` means "any character which is not a lowercase letter". Of couse, this does not include just uppercase letters, but punctuation marks, numbers and many special symbols too. The negation idea works with script names too, so `\P{Greek}` means "anything which is not in the Greek alphabet".

Composite characters

Another feature of Unicode is that of combining characters. This allows us to add diacritical marks by specifying the base character, and then the code for the required mark. So, for example, a letter U with an umlaut accent would be specified with the codes 85 followed by 168. However, as is often the case with Western European languages, it is also possible to use the single code 220. If you have to handle such characters in a regular expression, then the `\X` option is useful. It matches a base character followed by any number of optional combining characters, so it works regardless of whether the base character is written on its own, or with the combining character too.

Finally, with so many more characters than we can possibly enter on our keyboards, it would be helpful to be able to match a specific symbol by its code number. The `\x` option followed by a hexadecimal number allows us to do just that. This is also useful for finding control characters which cannot easily be included in a string. So `\x0A` matches linefeeds, and `\x{221A}` finds the square root symbol. Note that the braces are needed if the code number is more then two digits long.

Limitations

Finally, beware that while PCRE does offer facilities to handle Unicode, for reasons of speed the characters recognized by the familiar classes such as `\s` or `\w` do not expand accordingly. This includes the otherwise useful `\b` option which matches at the start and end of words. Like `\w` it

fails to recognize non-Roman scripts, and therefore cannot always recognize the begining of a new word.

The above has provided a summary of the facilities specific to Unicode text in regular expressions. This article will now continue to look at some more advanced features which add further power to regular expressions.

Backreferences

Recall from the first part of this series that we have already seen how to match an HTML tag. Now suppose we want to find any occurrences of one tag followed by its corresponding closing tag. On its own, a closing tag could be matched with `</\w+>`. However, if we combined this with the expression for an opening tag, it would find *any* opening and closing tag, not only matched pairs.

To limit ourselves to matching pairs we need to be able to to say something like "find the same text that you matched earlier". This can be achieved by using backreferences. Remember also that by enclosing part of an expression in parentheses you can capture the matching text for use later. Combining this with a backreference will enable us to match an opening tag, and then refer back to this to find the corresponding closing tag.

So by using `<(\w+)>` to match the opening tag, we can save the tag name for use later. If this finds the HTML `<font>` tag, then the brackets will capture the "`font`" part which we can refer to again using the backreference `\1` *[a numeral 1 —Ed]*. Once this has been done, the closing tag can be matched with `</\1>`. In this case the `\1` means "the text matched by the first set of brackets". It is also possible to write `\2` for the text matched in the second set of brackets and so on. Putting all this together gives the following expression to find a matching pair of HTML tags:

```
<(\w+)>.+</\1>
```

Double entendre

We've now seen that the round brackets have two possible meanings: either to group things together, or to capture a section of matched text for use later. In many cases this isn't a problem – after all, you don't have to use the associated backreference if you don't need to. However, it can cause some confusion when you look at your regular expression some time later.

More seriously, it also carries a small performance penalty, because the regular-expression algorithm has to remember these short sections of text. In a complicated expression this can slow things down, so there is also an option to use "non-capturing" parentheses if you don't need to refer back. This is effected by replacing the normal opening bracket by the sequence `(?:` while the closing bracket is left unchanged.

To see an example of this, return to the example of matching hexadecimal numbers, as part of an assignment to a variable. The number itself (after the equals sign) may begin with either `0x` in C programs or `&` in Basic, but really the number or the variable name is the interesting thing. Using non-capturing parentheses allows us to group the two possibilities together without the need to extract the substring or refer back to it. The complete expression is:

```
(\w+)=(?:0x|&)[0-9A-F]+
```

If you try this, you'll see only the first set of brackets captures the variable name, and the second set only groups the two alternatives.

Look-ahead

Thus far, we've seen that many

features in regular expressions have a parallel in programming lanuages, like alternation and repetition, for example. To complete the analogy there needs to be a parallel to the conditional statement. This is provided by the concept of "look-ahead", which in effect allows us to give instructions like "if the specified substring matches at this point, then try to match the following". To give a simple example, suppose that I wanted to look for the first three letters of my name. However, because I don't abbreviate my name to Nick, I'm not interested in that instance. Using look-ahead I can say "Match *Nic* only when it occurs as part of *Nicholas*". This is written as: `(?=Nicholas)Nic`.

This is perhaps a rather contrived example, but it does illustrate the point. Similarly, notice that the look-ahead does not consume any text, rather like the word-boundary metacharacters. So the example only checked to see if my name would match, and then proceeded to match Nic. This means that exactly the same effect could be achieved by writing `Nic(?=holas)`. In either case, it means that a subsequent part of the expression is free to match from the *h*.

Erratum

In the previous part of this series (Archive 23:10), I gave an example of using a regular expression to match internet domain names. This has caused some confusion, so may I make it clear that the example was intended only as an academic exercise.

As I hinted, the expression presented fails to catch some quite obvious cases. It could be improved upon with a much more complicated expression.

I failed to state, though, that it is impossible to validate domain names with a regular expression. There are simply too many possibilities, and the list is constantly expanding!

Another important point is that there are often limitations on the substring in the brackets. In Perl regular expressions, for example, the substring cannot be of variable length, so that you could not write `(?=the|then)` or even `(?=then?)`. PCRE, however, does allow the former, although the latter is still not permitted.

Look-behind

Having introduced the idea of look-ahead, it is now possible to mention that the idea works backwards too, when it is, unsurprisingly, called look-behind. This is occasionally useful if you want to match a string only when it is preceded by a particular substring. For example, the expression `(?<=\/\*)return` matches the word *return* only when it comes after a C-style comment delimiter. Run this over the text of a program and you'll find all comments explaining the return value of a function, but not the actual instruction. In this case, the look-behind isn't strictly necessary unless, of course, the comment delimiter has already been matched by a previous part of the expression.

Finally, both look-ahead and -behind can be negated. This works as before, but checks that the substring does not appear. Thus the expression `(?<![A-Z])\.\s` finds all instances of a full-stop and a space which do not follow capital letters. Essentially this usefully eliminates all cases where the full-stop occurs as part of an abbreviation. As an alternative, a negated character class could have been used instead.

Optimizing it

The above has introduced some advanced techniques, and by now you will have some idea of just how complicated regular expressions can

get. Although the simple examples shown here do not take an appreciable time to match, repeat them several times over, or try a complicated expression, and the running time can add up. If this is the case, a number of optimizations can be applied to make the expression run as fast as possible.

• If possible, use the start- and end-of-line anchors, `^` and `$`, which constrain your expression to match only at the beginning or end of a line. The word-boundary anchor, `\b`, can also speed things up.

• Try to include as much literal text as possible at the beginning of the expression. This works like the anchor above, but is more general. The technique allows the algorithm to quickly eliminate those locations where the expression cannot match. This can also work with alternations too, so `this|that` can be rewritten as `th(is|at)`. Although the two expressions are the same, the latter is often faster.

• Take special care with quantifiers. Sometimes it can be faster to use a quantifer rather than a repeated character, so `\d{3}` is often faster than `\d\d\d`. Conversely if we were matching the letter *d* rather than "any digit", the speed advantage disappears. Similarly, `\d\d*` is often faster than `\d+`.

• Use a character class rather than alternation. Thus `[abc]` is quicker than `a|b|c`. Of course this trick works only for a choice between single characters, but long alternations are also best avoided: instead, attempt separate matches with each of the alternatives. Where you can't reasonably avoid an alternation, try to put the most common case first.

• Omit unnecessary brackets: don't use a character class for a single letter, for example. This is an obvious example, but it can happen unintentionally if you start modifying an existing expression. Likewise, the brackets in expressions such as `(.)*` are unnecessary, and prevent the algorithm from applying any internal optimizations it may have. Also, where parentheses are necessary, try to use the non-capturing versions.

• Finally, remember that every time you use an alternation or a quantifier you introduce a loop. Nested loops are sometimes useful, but they can take a long time to complete. In particular, patterns like `(special|normal)*` are often very slow. It is possible to "unroll" the loop: rewrite the pattern as `normal*(special normal*)*`, which runs much quicker. Rather than examine specific cases here, I have included an example with this article.

Files: see p.2

More on the web

Here are two websites that give tutorials about regular expressions, as well as further examples and a library of them:

- *regular-expressions.info*
- *regexlib.com*

Using regular expressions in your own C programs

This article so far has discussed most aspects of constructing a regular expression. For those using existing software tools, this will be sufficient. However, part of the power of regular expressions is the ability to automate text-processing tasks in your own programs. The following section will explain how to use the PCRE library functions.

First, note that PCRE is a linkable C library, so at present it can only be used from that programming language, and these notes will assume familiarity with C. For a more thorough example of PCRE in use, the source code of my Regex tool is a

good starting point. All of the function prototypes and definitions are in h.pcre, so your code will need to include the line

```
#include "pcre.h"
```

The first step is to "compile" your pattern into PCRE's internal representation. The necessary function is:

```
pcre *pcre_compile(const char *pattern,
int options, const char **errptr, int
*erroffset,const unsigned char
*tableptr);
```

This takes a pointer to your pattern and returns a pointer to the compiled representation. The memory for this is automatically obtained via `malloc`, so you should free it when you've finished with the pattern by calling `pcre_free`. If there are any errors in the pattern, then a null pointer is returned, while `errptr` and `erroffset` are set up with a description and the offset within the pattern.

The options word allows you to select, among other things, the newline convention, case-insensitive matching, and Unicode support. The definitions for these are found in `h.pcre` as well. The following fragment of code is sufficient to compile a simple pattern:

```
#include <stdio.h>
#include <string.h>
#include "pcre.h"

char patt[80];
cont char *errstr;
pcre *regexp;
int opts, errpos;

strcpy(patt, "pattern");
/* case insensitive and UTF8 */
opts=PCRE_CASELESS|PCRE_UTF8;
regexp=pcre_compile(patt, opts, &errstr,
&errpos, NULL);
if(!regexp)
printf("Error compiling pattern: %s at
byte %d\n", errstr, errpos);
```

Optimize

Having compiled the pattern, it is possible to optimize it with `pcre_study`, although this incurs a further processing overhead. This may be worthwhile for particularly complicated patterns, or if you intend to match the same pattern against several strings. Whether or not you choose to optimize the pattern, it is now possible to match it against the subject string. You can do it using this function:

```
int pcre_exec(const pcre *code, const
pcre_extra *extra, const char *subject,
int length, int startoffset, int options,
int *ovector, int ovecsize);
```

This takes pointers to the compiled pattern, any extra data produced by optimization, and the subject string. It returns the number of captured substrings, or a negative value in case of an error. The subject string may contain null bytes, so it is also necessary to give the length of this string.

Information on the substring that matched, and any captures, are returned in the `ovector` array. This needs to be large enough to accommodate three integers each for the substring and any captures. It may be useful to call `pcre_fullinfo` to get the necessary size of the `ovector` array. You can either use `pcre_malloc` to obtain the memory for `ovector` or, if the size is known in advance, declare a static array.

`pcre_exec` returns the number of pairs of entries used in `ovector`. The first pair is the offsets to the start and end of the string matched by the entire pattern; the second pair represents the first captured substring and so on. If the pattern did not match, then `pcre_exec` returns −1.

All this is illustrated in the following fragment to set up memory, match the compiled pattern and display the result:

```
char subj[80], ssub[80];
```

```
 int subjlen, veclen, rc, *ovec;

 strcpy(subj, "subject");
 subjlen=strlen(subj);
 /* allocate memory */
 rc=pcre_fullinfo(regexp, NULL,
PCRE_INFO_CAPTURECOUNT, &veclen);
 veclen=(veclen+1)*3*sizeof(int);
 ovec=pcre_malloc(veclen);
 /* match regexp */
 rc=pcre_exec(regexp, NULL, subj,
subjlen, 0, 0, pcreout, veclen);
 if(rc==-1)
 printf("No match!\n");
 else {
 /* print matching substring */
 pcre_copy_substring(subj, ovec, veclen,
0, ssub, 80);
 printf("Matched %s at byte %d\n", ssub,
ovec[0]);
 }
```

Compile and link

Finally, once you have written your program, you will need to compile and link it with the PCRE library. I normally keep PCRE along with the other libraries on the system and set `PCRE$path` to its location. Once this has been done, you can compile your program with the following command:

```
cc -c -IC:,PCRE: -o o.regex c.regex
```

To produce the final executable program, link it with PCRE and any other libraries you need, including the shared C library:

```
link -aif -Output regex o.regex
PCRE:o.pcrelib C:o.stubs
```

Summing up

This article completes my series on regular expressions by looking at some of the more advanced features, and summarizing how you can use these facilities in your own programs. I hope it has encouraged you to experiment, both with regular expressions, and with the PCRE library itself.

As with part 1, I have supplied the Archive website with a number of examples you can try using my Regex program. Go to *www.cl.cam.ac.uk/~ncc25/pcre_riscos.zip* to download the source code of the Regex program and a linkable version of the PCRE library.

Nicolas Cutler *ncc25@cam.ac.uk*

Parallel processing in action: Nicholas and a friend competing in the Great Ouse marathon. Being at the front – and facing backwards – Nicholas had the additional responsibility of steering!

Christmas cheer: Our tame garden robin

Barry and Ann Jerome, about their Christmas card (in colour on back cover):

This robin has been a constant companion in the garden here in Bishops Waltham, Southampton, for over a year. He first started to follow us when we were gardening, coming very close to pick up grubs. In the spring the robin had a mate and raised a family of young. We started to put out live mealworms and soon after he brought the family to show us and to feed on the mealworms.

About this time we noticed he appeared to be be suffering from a bird illness called gapeworm. We searched the web and found a herbal remedy: rosemary. We mixed ground rosemary into the birdfood, and he soon began to recover.

We have continued to feed him with dried mealworms and grated cheese. He now appears when we whistle and whistles back to us when we feed him.