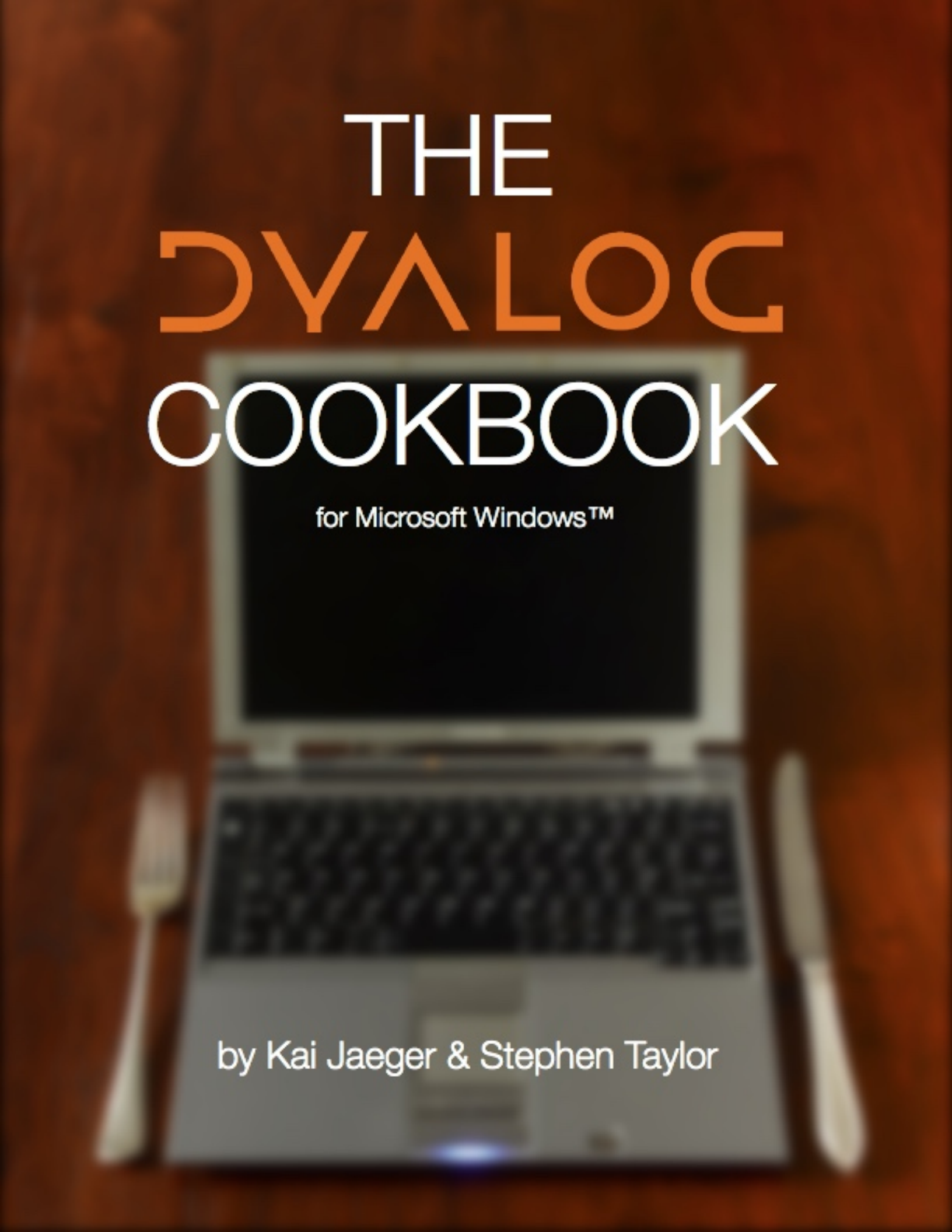
THE DYALOG COOKBOOK

for Microsoft Windows™



by Kai Jaeger & Stephen Taylor

The Dyalog Cookbook

Kai Jaeger and Stephen Taylor

This book is for sale at http://leanpub.com/thedyalogcookbook

This version was published on 2017-08-11

With grateful thanks to Kenneth E. Iverson (1920-2004), for a language we can think in, and to his collaborators and followers who make it work.

Contents

1. Regular expressions with Dyalog

1.1 To whom it concerns

If you are fluent in regular expressions then you may skip this chapter except perhaps “Analyzing APL ocde” which might tell you about a particular strength of Dyalog’s implementation of regular expressions.

If you are not fluent in regular expression but heavily involved into number crunching without any memory of ever having scanned strings for certain patterns then there is no point in looking into regular expressions because you just don’t need them. Let’s be clear, regular expressions are an extremely powerful tool, but the level of abstraction is really high. You will find it hard to master them without using them regularly. To rephrase it, if you need to find a pattern in a string twice a year you are probably better off finding an expert on the matter you can ask for advice.

Having said this it is amazing that many APLers do not realize how often they actually to search for strings and patterns.

1.2 Overview

In this chapter we take the approach to explain regular expressions purely by example. The examples start simple and grow complex. Along the line we introduce more features of regular expressions in general and Dyalog’s implementation in particular. Your best strategy is to read the following stuff from start to end. It will introduce you to the basic concepts and provide you with the necessary knowledge to become a keen amateur. From there constant usage of regular expressions and the Internet will convert you into an expert, though it will take a bit of time and effort. Be assured that it will be well invested time.

This chapter is by no means a comprehensive introduction to regular expressions, but it should get you to a point where you can take advantage of examples, documents and books that are not addressing Dyalog’s implementation.

Note that we explain the syntax of ⎕R and ⍠R separately from the main text. That makes it easy to ignore those bits in case you are already familiar with the syntax.

Despite the name of the book you will find only very few recipes for real-world problems in this chapter. The problems provided are used as vehicles to introduce the main features of regular expressions.

1.3 What are regular expressions?

Regular expressions allow you to find the position of a search string in another string. They also allow you to replace a string by another one.

1.4 Background

Dyalog is using the PCRE implementation of regular expressions. This library attempts to stay as close as possible to the Perl 5 implementation. There are many other implementations available, and they all differ more or less. Therefore it is important to know what kind of engine you are actually using when you do RegExes from within Dyalog. Dyalog 16.0 uses PCRE version 8. Note that PCRE is considered one the most complete and powerful implementations of regular expressions.

1.5 Examples

Example 1 - search a string in a string

```
      ⍴'notfound' ⎕s 0⊣ 'the cat sat on the medallion'
0
      'cat' ⎕s 0⊣ 'the cat sat on the medallion'
 4
```

The right operand and the right argument

`⎕S` is, like `⎕R`, an *operator*. An operator takes either just a left operand (monadic) or a left and right operand (dyadic) and forms a so-called derived function. For example, the operator `/` when fed with a left operand `+` forms the derived function "sum".

In the example the `0` is the right operand. With `⎕S` the right operand can be one to many of 0, 1, 2 and 3 (those are called transformation codes) or a user defined function which is explained later.

- `0` stands for: offset from the start of the line to the start of the match.
- `1` stands for: length of the match.

The transformation codes `2` and `3` will be discussed later.

The right argument provided to the derived function is the string `'the cat sat on the medallion'`. The operand and the string are separated by the `⊣` function. Instead we could have used parenthesis with exactly the same result: `('notfound' ⎕s 0) 'the cat sat on the medallion'`.

In the first expression the result is empty because the string `notfound` was not found. In the second expression `cat` was actually found. That means that we could say:

```
      'cat' {⍵↓⍨⍺ ⎕s 0 ⊣ ⍵}'the cat sat on the medallion'
cat sat on the medallion
```

That was easy, wasn't it! Obviously regular expressions are nothing to be afraid off. Let's look at another example: find out what's between double quotes. First attempt:

```
      '"' ⎕S 0 ⊣ 'He said "Yes"'
 8 12
```

That gives us the offset of the two double quotes, but what if we want to have the offset and the length of any string found *between* — and including — the double quotes? For that we need to introduce the *meta character* dot (`.`) which has a special meaning in a regular expression: it represents *any* character (not strictly true but we will soon discuss the exception, the NewLine character).

Meta Characters

Meta characters, sometimes called special characters, are those characters that have a special meaning in a regular expression. In order to really master regular expressions you have to know *all of them.* That will not only enable you to use them properly, it will also prevent you from advertising yourself as a non-skilled RegEx user: those tend to escape pretty much everything that is not a letter or a digit because they don't know what they are doing. Even if you don't care, unnecessary escaping also reduces readability significantly; you really should only escape the characters that *need* to be escaped.

Note that if you want a meta character to be taken literally (like searching for a dot) then you have to escape the meta character by a backslash (\) which therefore is itself a meta character. The expression `'"\.\\"'` would search for double-quote followed by a single dot followed by a single backslash followed by a double-quote.

So we try:

```
      '"."' ⎕S 0 1 ⊣ 'He said "Yes"'
```

Opps - no hit.

In order to understand this we have to know *exactly* what the regular expression engine did:

1. It starts at the beginning of the string; that is actually one to the left of the initial "t"! That position can only match to the meta character `^` which represents the start of a line.
2. If there is no match the engine forgets about it and moves one character forward. This is called "consuming" the position the engine has looked at.
3. It then tries to match `"` to `H`. Since there is no match either…
4. It carries on until it arrives at the `"`. Now there is a match!
5. The engine now tries to match the `.` against the `Y`. Since the`.` matches *any* character this is a match, too.
6. It then moves forward one more character and tries to match the `"` with `e` - that's *not* a match.
7. The engine forgets what it has done, goes back to where it started from (that was the `"`) and moves one character forward.
8. It now tries to match the `"` with the `Y` …

You can now see why it does not report any hit - it would work on `'He said "Y"'` or more generally, on any single character that is embraced by two double quotes.

What we need is a way to tell the engine that it should try to match the `.` more than once. There is a general way of doing this and two shortcuts that cover most cases. For example, in order to match a minimum of one to a maximum of three underscore characters:

```
      '_{1,3}' ⎕S 0 ⊢'_one__two___three____four'
0 4 9 17 20
```

It is actually easier to check the result by replacing the hits with something outstanding:

```
      '_{1,3}' ⎕R '□' ⊢'_one__two___three____four'
□one□two□three□□four
      '_{2,3}' ⎕R '□' ⊢'_one__two___three____four'
_one□two□three□_four
```

The right operand

`⎕R` takes one or more replace strings *or* a user defined function (discussed later) as the right operand; you cannot mix replace strings with user defined functions.

What's between the curlies (`{}`) – which are meta characters as well – defines how many are required as minimum and maximum. This is called a quantifier.

- `{x,y}` means a min of x and a max of y.
- `{,y}` is the same as `{0,y}`.
- `{x,}` means a min of x with no max limit.
- `{x}` means exactly x.
- The star (`*`) is a shortcut for `{0,y}` and `{,y}`.
- The plus (`+`) is a shortcut for `{1,}`.

Therefore:

```
      '".*"' ⎕R '□' ⊣ 'He said "Yes"!'
He said □!
```

Seems to work perfectly, right? Well, keep reading:

```
      '".*"' ⎕R '⍠' ⊣ 'He said "Yes" and "No"!'
He said ⍠!
```

Just one hit, and that hit spans `"Yes" and "No"`?! That's because by default the engine is *greedy* as opposed to *lazy*: it carries on and tries to match the `.` against as many characters as it can. That means that in our example it will stop only after it reached the end of the line, because all characters found are a match. It would then go back until it finds a `"` (coming from the right!) and then stop because it's done the job.

The same is true for the `{x,y}` quantifiers: by default they are all *greedy* rather than *lazy*, so repeating `y` times is tried before reducing the repetition to `x` times.

What we need instead is a *lazy* search, therefore:. We can change the default by specifying the "Greedy" option:

```
      '".*"' ⎕R '⍠' ⍠('Greedy' 0) ⊣ 'He said "Yes" and "No"!'
He said ⍠ and ⍠!
```

Options

Between the right operand and the right argument you may specify options. They are marked by the `⍠` operator. These are the options available:

IC, Mode, DotAll, EOL, ML, Greedy, OM, InEnc, OutEnc, Enc, ResultText, UCP

Note that "IC" is the principle option. That means that if no other option needs to be specified you can omit the "IC"; this would do: `⎕S 0 ⍠ 1 ⊢ 'whatever'`. "IC" stands for "Ignore Case". A 1 would make a search pattern case insensitive. The default is 0.

In this chapter we won't discuss all these options but "IC", "Mode", "DotAll", "Greedy" and "UCP". For the others refer to the help page of `⎕R`.

Let's repeat our findings because this is so important:

- Greedy means match *longest* possible string.
- Lazy means match *shortest* possible string.

Garbage in, garbage out

Let's modify the input string:

```
      is←'He said "Yes"" and "No"'  ⍝ define "input string"
```

There are *two* double quotes after the word "Yes"; that seems to be a typo. Watch what our RegEx is making of this:

```
      '".*"' ⎕R '⍠' ⍠('Greedy' 0) ⊣ is
He said ⍠⍠No"
```

This example highlights a potential problem with input strings: many regular expressions work perfectly well as long as the input string is, well, let's say syntactically correct. That's (among other reasons) why regular expressions are not recommended for processing HTML because HTML is very often full of syntactical errors. However, if you can be certain that the HTML you have to deal with *is* syntactically correct *and* the piece of HTML is short then there is nothing wrong with using regular expressions to process it.

Regular expressions and HTML

There are claims that you *cannot* parse HTML with regular expressions because a regular expression engine is a Finite Automata while HTML can be nested indefinitely. This is partly wrong and partly misleading.

- It is wrong because today's regular expression engines come with features (back referencing) that are clearly beyond the feature set of a Finite Automata.
- It is misleading because it ignores that regular expression engines are perfectly capable of processing small pieces of HTML that are known to be syntactically correct.

By now we've met quite a number of meta characters; how many do we have to deal with? Well, quite a lot:

	Meta character	Symbol		Meaning
1.	Backslash	`\`	✓	Escape character
2.	Caret	`^`	✓	Start of line
3.	Dollar sign	`$`		End of line
4.	Period or dot	`.`	✓	Any character but NewLine
5.	Pipe symbol	`|`		Logical "OR"
6.	Question mark	`?`		Extends meaning of `(`; `0` or `1` quantifier (=optional); makes it lazy
7.	Asterisk or star	`*`	✓	Repeat `0` to many times
8.	Plus sign	`+`	✓	Repeat `1` to many times
9.	Opening parenthesis	`(`		Start sub pattern
10.	Closing parenthesis	`)`		End sub pattern
11.	Opening square bracket	`[`	✓	Start character set
12.	Opening curly brace	`{`	✓	Start min/max quantifier

By now we have already discussed six of them; they carry a check mark.

Note that both `}` and `]` are considered meta characters only after an opening `{` or `[`. Without the opening counterpart they are taken literally; that's why they did not make it into the list of meta characters.

Example 2 - digits in a string

Let's assume we want to match all digits in a string:

```
      '[0123456789]'⎕R '⍰' ⍠ 'It''s 23.45 plus 99.12.'
It's ⍰⍰.⍰⍰ plus ⍰⍰.⍰⍰.
```

Everything between the `[` and the `]` is treated as a simple character - with a few exceptions we'll soon discuss. That makes both `[` and `]` meta characters.

The same but shorter:

```
      '[0-9]'⎕R '⍰' ⍠ 'It''s 23.45 plus 99.12.'
It's ⍰⍰.⍰⍰ plus ⍰⍰.⍰⍰.
```

Note that the hyphen is treated as a meta character here: it means "all digits from 0 to 9".

Even shorter:

```
      '\d'⎕R '⍰' ⍠ 'It''s 23.45 plus 99.12.'
It's ⍰⍰.⍰⍰ plus ⍰⍰.⍰⍰.
```

Note that the meta character backslash (`\`) is used for two different purposes:

- To escape any of the RegEx meta characters so that they are stripped of their special meaning and taken literally.
- To give the next character a special meaning in case it is an ordinary ASCII letter.

So `\*` takes away the special meaning from the `*` while `\d` gives the `d` the special meaning "all digits".

We take the opportunity to add the dot (`.`) and the hyphen (`-`) to the character class. Note that the hyphen is not escaped; from the context the regular expression engine can work out that here the hyphen cannot mean from-to, so it is taken literally.

```
      '[\d.-]'⎕R '⍰' ⍠ 'It''s 23.45 plus -99.12.'
It's ⍰⍰⍰⍰⍰ plus ⍰⍰⍰⍰⍰⍰⍰
```

Here we have another problem: we want the dot only to be a match when there is a digit to both the left and the right of the dot. Our search pattern is not dealing with this, therefore the trailing `.` is a match. We will tackle this problem soon with look-ahead and look-behind.

Character classes work for letters as well:

```
      '[a-zA-Z]'⎕R '⍰' ⍠'It''s 23.45 plus 99.12.'
⍰⍰'⍰ 23.45 ⍰⍰⍰⍰ 99.12.
```

We can negate with `^` right after the opening `[`:

```
    '[^a-zA-Z]'⎕R '⍈' ⍤1⊢'It''s 23.45 plus 99.12.'
It⍈s⍈⍈⍈⍈⍈⍈⍈plus⍈⍈⍈⍈⍈⍈⍈
```

Notes:

- The `^` has a different meaning in a character class definition. Outside character classes it stands for "Start of line" as mentioned earlier.
- Only at the beginning of a character class definition has the caret the meaning "negate". Therefore you could also say that `[^` means "negate" while, say, `[1^2]` means "Match for one of these three characters: `1^2`.
- For APLers the caret is a bit tricky because it can easily be confused with the logical AND (`∧`) function. Only next to each other it becomes apparent what it what: `^∧`: the caret is a bit higher than the logical AND, and a bit smaller as well.

Many problems can be solved in more than one way with regular expressions. For example, our earlier problem to find everything between (and including) double quotes can be solved with this expression as well: `'"[^"]*"' ⎕R '⍈' ⍤ 'He said "Yes", she said "No".'`. It matches a double quote, then as many characters as possible that are *not* a double quote and finally the closing double quote. This expression has the advantage of not depending on the setting of the "Greedy" option.

Negate with digits and dots:

```
      '[^.0-9]'⎕R '⍈' ⍤1⊢'It''s 23.45 plus 99.12.'
⍈⍈⍈⍈⍈23.45⍈⍈⍈⍈⍈⍈99.12.
```

Want to search for "gray" and "grey" in a document?

```
      'gr[a|e]y'⎕S 0⍤1⊢'Americans spell it "gray" and Brits "grey".'
20 37
      'gr[a|e]y'⎕R '⍈' ⍤1⊢'Americans spell it "gray" and Brits "grey".'
Americans spell it "⍈" and Brits "⍈".
```

So the pipe symbol (`|`) has a special meaning inside a character class: it means logical "OR".

Note that there are only a few meta characters inside character classes:

Meta character	Symbol	Meaning
Closing bracket	`]`	
Backslash	`\`	Escape next character
Caret	`^`	Negate the character class
Hyphen	`-`	From-to

We already worked out that the engine is smart enough to take a hyphen literally when it makes an appearance

somewhere where it cannot mean from-to: the beginning and the end of a character class. Similarly the caret (^) character can only negate a character class as a whole, when it follows the opening square bracket ([^). If the caret is specified elsewhere it is taken literally. Therefore the expression `[0-9^1]` does *not* mean "all digits but 1, it means "all digits plus the caret character plus a 1".

Finding 0 to 3 white space characters followed by an ASCII letter at the beginning of a line

```
      {'^\s{0,3}[a-zA-Z]' ⎕R '⍟' ⊣ ⍵}¨'Zero' ' One' '  Two' '   Three' '    four'
⍟ero  ⍟ne  ⍟wo  ⍟hree      four
```

`\s` escapes the ASCII letter "s", meaning that the "s" takes on a special meaning: `\s` stands for "any whitespace". That is at the very least the space character (`⎕UCS 32`) and the tab character (`⎕UCS 9`). There are two options (the stuff that can be set with the `⍠` operator) that influence which characters qualifiy as white space:

- "Mode" (discussed soon).
- "UCP".

Analyzing APL code

Let's assume that you want to investigate APL code for a variable name `foo` but you want text and comments to be ignored. This is our input string:

```
      is←'a←1 ⋄ foo←1 2 ⋄ txt←''The ⍝ marks a comment; foo'' ⍝ set up vars a, foo, txt'
```

We want `foo←1 2` to be found/changed while the text and the comment shall be ignored/remain unchanged. The problem is aggravated by the fact that the text contains a `⍝` symbol.

The naive approach does not work:

```
      'foo' ⎕R '⍟' ⊣ is
a←1 ⋄ ⍟←1 2 ⋄ txt←'The ⍝ marks a comment; ⍟' ⍝ set up vars a, ⍟, txt
```

Dyalog's implementation of regular expressions offers an elegant solution to the problem:

```
      '''.*''' '⍝.*$' 'foo'⎕R(,¨'&&⍟')⍠('Greedy' 0)⊣is
a←1 ⋄ ⍟←1 2 ⋄ txt←'The ⍝ marks a comment; foo' ⍝ set up vars a, foo, txt
```

This needs some explanation:

1. `'''.*'''` catches all text, and for that text `&` is specified as replacement. Now `&` stands for the matching text, therefore nothing will change at all but the matching text *won't participate in any further actions!* In other words: everything between quotes is left alone.

2. `'⍝.*$'` catches everything from a lamp (`⍝`) to the end of the line (`$`) and replaces it by itself (`&`). Again nothing changes but the comment will not be affected by anything that follows. Since the first expression has already masked eveything within (and including) quotes the first `⍝` does not cause problems.
3. Finally `foo` catches the string “foo” in the remaining part, and that is what we are interested in.

As a result `foo` is found within the code but neither within the text nor as part of the comment.

As far as we know this feature is specific to Dyalog, but then we have limited experience with other regular expression engines.

Note that the `,¨` in `,¨'&&⍈'` is essential because otherwise ….⍝TODO⍝ Bug report <01406>

⍰TODO⍰

⍰A> ### Greedy and lazy ⍰A> ⍰A> Note that using the option (`⍈('Greedy' 0)`) has a disadvantage: it makes the *whole search pattern* lazy. There might be cases when you want part of your search pattern to be lazy and other parts greedy. Luckily this can be achieved with the meta character question mark (`?`): ⍰A> ⍰A> ~~⍰A> ‘”.*?”⍰R‘⍰’ ⊣ is ⍰A> He said ⍰ and ⍰⍰A>~~ ∼ ⍰A> ⍰A> Since “Greedy” is the engine’s default you need to specify the `?` only for those parts of your search pattern you want to be lazy.

Our search pattern is still not perfect since it would work on `boofoogoo` as well:

```
      '''.*''' '⍝.*$' 'foo'⎕R(,¨'&&⍈')⍈('Greedy' 0)⊣'This boofoogoo is found as well'
This boo⍈goo is found as well
```

To solve this we need to introduce look-ahead and look-behind. The names make it pretty obvious what they do. We want to emphasize that all matching attempts we’ve introduced so far have been “consuming”. Look-ahead as well as look-behind are *not* consuming. That means that no matter whether they are successful or not they won’t change the position the engine is currently investigating. They are also called zero-length assertions.

However, before we tackle our problem we need to introduce the concept of both word boundaries and anchors. We’ve already met one anchor: the caret (`^`), which matches the beginning of a line. There is also the dollar (`$`) which matches the end of a line. And there is `\b` which matches a “word boundary”. All these characters are zero-length matches.

To put it simply, `\b` allows you to perform a “whole word only” search. Prior to version 8 of PCRE (and 16.0 of Dyalog) this was true only for ASCII characters. Now you can set the “UCP” option to 1 if you want Unicode characters to be taken into account as well:

```
       ⍴'\bger\b'⎕S 0 ⊣'Kai Jägerßabc'
1
      ⍴'\bger\b'⎕S 0 ⍈('UCP' 1)⊣'Kai Jägerßabc'
0
```

The following uses both, look-ahead and look-behind for making sure that “ger” stands on its own:

```
      ss←'ger :ger ger! Jaeger Jägerßabc'
      '(?<=\b)ger(?=\b)'⎕R'⍈' ⍝ ss
⍈ :⍈ ⍈! Jaeger Jä⍈ßabc
```

Both look-ahead and look behind start with `(?`. A look behind then needs a `<` while the look-ahead doesn't. Both then need either a `=` for "equal" or a `!` for not equal" followed by the search token and finally a closing `)`. Hence `(?<=\b)` for the look-behind and `(?=\b)` for the look-ahead.

What the engine does:

- Since the search pattern starts with a look-behind the engine checks whether there is a word boundary *to the left of the current position.*
- It's the beginning of the line, so that's successful, and the engine then checks whether the current position matches a "g".
- That's successful, so the engine moves forward and tries to match the "e" with the current position.
- That's successful too, so the engine moves forward again and tries to match the "r" with the current position.
- That a match as well, so the engine performs a look-ahead: *without moving forward* it tries the match the character *after* the current one to be a word boundary. A space character qualifies as a word boundary.
- Thats a success, too, so the "ger" is replaced by a single ⍈.

What is important to realize is that the current position does not change when a look-behind or a look-ahead is performed (though the current position might change as a result of a failing test); that's why they are called zero-length assertions.

In the same way the following two appearances of "ger" are replaced by ⍈ because both `!` and `.` qualify as word boundaries as well. The "ger" in "Jaeger" was not replaced beecaus the look-behind failed: "e" is not a word boundary. Same for the last one: neither "ä" nor "ß" qualify as word boundaries because they are non-ASCII characters.

That makes word boundaries pretty useless for other languages then English! Luckily with version 16.0 Dyalog was able to start using version 8 of the PCRE engine which now supports the Unicode definition of word boundaries. To take advantage of this feature we have to specify the "UCP" option:

```
      ss←'ger :ger ger! Jaeger Jägerßabc'
      '(?<=\b)ger(?=\b)'⎕R'⍈' ⍈('UCP' 1) ⍝ ss
⍈ :⍈ ⍈! Jaeger Jägerßabc
```

Now both "ä" and "ß" qualify as word boundaries.

We can use look-ahead and look-behind to solve a problem we did run into with numbers. This did not really work because *all* dots got replaced when we wanted only those with digits to the right and the left being a match:

```
      '[\d.¯-]'⎕R'⍈'⊣'It''s 23.45 plus 99.12.'
It's ⍈⍈⍈⍈⍈ plus ⍈⍈⍈⍈⍈⍈
```

We don't want the last dot to be a match:

```
      '\d' '(?<=\d).(?=\d)'⎕R'⍈'⊣'It''s 23.45 plus 99.12.'
It's ⍈⍈⍈⍈⍈ plus ⍈⍈⍈⍈⍈.
```

That works! We use two expression here: first we look for all digits and then we look for dots that have a digits to their right and their left. However, in case you need ⎕S to return the start and the length of any matches then the result is unlikely to be what you are after:

```
      '\d' '(?<=\d).(?=\d)'⎕S 0 1 ⊣'It''s 23.45 plus 99.12.'
 5 1  6 1  7 1  8 1  9 1  16 1  17 1  18 1  19 1  20 1
```

We need an expression that identifies any vector of digits as one unit, no matter whether there is a dot between the digits or not:

```
      '\d+(?<=\d).(?=\d)\d+' ⎕R '⍈' ⊣ 'It''s 23.45 plus 99.12.'
It's ⍈ plus ⍈.
      '\d+(?<=\d).(?=\d)\d+' ⎕S 0 1 ⊣ 'It''s 23.45 plus 99.12.'
 5 5  16 5
```

That's better.

Naturally you will need a way to *negate* a look-ahead and a look-behind. That can be achieved by using a `!` rather than a `=`.

Lets' repeat this. Assuming we look for "x" and "y":

```
      'x(?<=y)' ⎕R'⍈' ⊣ 'abxycxd' ⍝ Exchange all "x" when followed by a "y"
ab⍈ycxd
      'x(?!y)' ⎕R'⍈' ⊣ 'abxycxd' ⍝ Exchange all "x" when NOT followed by a "y"
abxyc⍈d
      '(?<=x)y' ⎕R'⍈' ⊣ 'abxycyd' ⍝ Exchange all "y" when preceeded by an "x"
abx⍈cyd
      '(?<!x)y' ⎕R'⍈' ⊣ 'abxycyd' ⍝ Exchange all "y" when NOT preceeded by an "x"
abxyc⍈d
```

1.6 Transformation function

Instead of providing a replace string one can also pass a function as operand to ⎕R. Our earlier example:

```
      is←'a←1 ⋄ foo←1 ⋄ txt←''text; foo'' ⍝ comment'
```

Let's replace just the variable name with something else with a transformation function:

```
      ∇test[⎕]∇
[0]   r←{x}test y
[1]   .

      '''.*''' 'A.*$' 'foo'⎕R  test⍤is
SYNTAX ERROR
test[1] .
       ^
      y.(⊃{⍵ (±⍵)}¨↓⎕nl 2 9)
 Block        a←1 ⋄ foo←1 ⋄ txt←'text; foo' A comment
 BlockNum                                          0
 Lengths                                           3
 Match                                           foo
 Names
 Offsets                                           6
 Pattern                                         foo
 PatternNum                                        2
 ReplaceMode                                       0
 TextOnly                                          0
```

The right argument

We modify `test` so that is leaves the text and the comment untouched:

```
      )reset
      ∇test[⎕]∇
[0]   r←{x}test y
[1]   :If ''''''≡2⍴¯1⍵y.Match
[2]       ⎕←r←y.Match
[3]   :ElseIf 'A'=1⍴y.Match
[4]       ⎕←r←y.Match
[5]   :Else
[6]       r←'⍞Hello world⍞'
[7]   :EndIf

      '''.*''' 'A.*$' 'foo'⎕R test⍤is
'text; foo'
A comment
a←1 ⋄ ⍞Hello world⍞←1 ⋄ txt←'text; foo' A comment
```

Since any match that starts and ends with a quote is text by definition the function returns those untouched. Anything that start with a lamp symbol is a comment, so they are returned untouched as well. That leaves the hits for the real variable names: they are exchanged against `⍞Hello world⍞`.

Naturally transformation functions gives you enourmes power: you can do whatever you like.

1.7 Document mode

So far we have specified just a simple string as input. We can however pass a vector of strings as well. Look at this example:

```
      input←'He said: "Yes, that might' 'well be right." She answered: "So be it!"'
```

It's not a bad idea to think of the two elements of the input vector as "blocks". Note that the first text spans over both blocks.

By default the search engine operates in "Line" mode. That means that each block is processed independently by the engine. Therefore you cannot search for `\r` in line mode: the search engine will never see them.

In mixed as well as document mode you *can* search for `\r` because all blocks are passed at once. Naturally this also requires more memory than line mode.

Let's do some tests:

```
      '".*"'⎕R'␣'␣('Greedy' 0)⊣input
 He said: "Yes, that might  well be right.␣So be it!"
      '".*"'⎕R'␣'␣('Greedy' 0)('Mode' 'M')('DotAll' 1)⊣input
He said: ␣ She answered: ␣
```

Note that in order to specify `('DotAll' 1)` it is necessary to set `('Mode' 'M')`. `('Mode' 'D')` would have worked in the same way. However, when it comes to `^` and `$` then it makes a big difference:

- In line mode (`('Mode' 'L')`) `^` finds the start of the line and `$` finds the end of the line.
- In mixed mode (`('Mode' 'M')`) `^` finds the start of each block and `$` finds the end of each block.
- In document mode (`('Mode' 'D')`) `^` finds the start of the document and `$` finds the end of the document.

Note that rather than specifying ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕

1.8 Misc

Tests

Given that complex regular expressions are hard to read and maintain you should document intensively. The best way to document them is to write exhaustive test cases. Therefore we highly recommond that you write test cases at least for the more complex regular expressions.

Performance

Don't expect regular expressions to be faster than a taylored APL solutions; instead expect them to be slightly slower.

However, many regular expressions like finding a simple string in another simple string or uppercasing or lowercasing characters are converted by the interpreter into a native (=faster) APL expression (`⍷` and (`⎕C 819`).

Helpful stuff

RegexBuddy

This is a software that helps interpreting (or building) regular expressions.

http://www.regular-expressions.info/tutorial.html

This is a web site that really goes into the details. It's from the author of RegExBuddy.

The web site also comes with detailed book reviews: http://www.regular-expressions.info/hipowls.html