

## **kSpam**

*- An open source Lotus Domino spam filter.*

# kSpam

– *An open source Lotus Domino spam filter.*

## Contents

1.	Overview.....	3
2.	The files.....	4
3.	Setup -Windows.....	5
4.	Standard configuration.....	6
5.	Bayesian setup and configuration.....	7
6.	Rules.....	8
	Rule Example: .....	8
7.	Limitations.....	9
8.	FAQs .....	10
	Appendix A – Rule examples .....	11
	Appendix B - Notes.ini variables .....	14
	Appendix C – Regular expression references.....	18
	PCRE REGULAR EXPRESSION DETAILS .....	18
	BACKSLASH .....	19
	CIRCUMFLEX AND DOLLAR.....	23
	FULL STOP (PERIOD, DOT).....	24
	MATCHING A SINGLE BYTE .....	25
	SQUARE BRACKETS.....	25
	POSIX CHARACTER CLASSES .....	26
	VERTICAL BAR .....	27
	INTERNAL OPTION SETTING.....	28
	SUBPATTERNS .....	29
	NAMED SUBPATTERNS .....	30
	REPETITION.....	30
	ATOMIC GROUPING AND POSSESSIVE QUANTIFIERS.....	33
	BACK REFERENCES.....	35
	ASSERTIONS .....	36
	CONDITIONAL SUBPATTERNS.....	39
	COMMENTS .....	40
	RECURSIVE PATTERNS.....	40
	SUBPATTERNS AS SUBROUTINES .....	42
	CALLOUTS .....	43

# kSpam

– *An open source Lotus Domino spam filter.*

## 1. Overview

kSpam is an open source spam filter for Lotus Domino, that includes rule based and Bayesian filtering.

Master Chef(s)

Tom Lyne

Nico Vis

Cooks on the team

Scott Ma

Hans-Georg Franke

Fabrice Verrac

Christian Brandlehner

# kSpam

– *An open source Lotus Domino spam filter.*

## 2. The files

kSpam.doc	This file
kSpam.pdf	This file in PDF format
kSpam.changelog.txt	The changelog.
kSpamCon.ntf	The configuration database template. (Required to create kSpamCon.nsf configuration database)
License.txt	GNU General Public License which the kSpam code is distributed under.
mailgood.ntf	The good mail database template. (Optional, for Bayesian operation)
mailspam.ntf	The spam mail database template. (Required to create mailspam.nsf database)
Version.txt	This kSpam version.
nspam.dll	Windows - The main program, a Domino extension manager addin. (Required for standard operation)
nbload.exe	Windows - The Bayesian statistical probability addin. (Optional, for Bayesian operation)
kspam	Linux - The main program, a Domino extension manager addin. (Required for standard operation)
libkspam.so	Linux - The Bayesian statistical probability addin. (Optional, for Bayesian operation)

# kSpam

– *An open source Lotus Domino spam filter.*

## 3. Setup – Windows and Linux

To setup kSpam follow these steps;

### **Only On Windows:**

1. Copy nspam.dll and nload.exe to the Domino program directory.
2. Add the following line to your notes.ini file

Extmgr\_addins=spam

If the line extmgr\_addins already appears in your notes.ini file append a comma and then the word "spam" to the end of the line, e.g.

Extmgr\_addins=av\_program,spam

### **Only On Linux:**

1. Copy kspam and libkspam.so to the Domino program directory (eg. /opt/ibm/lotus/notes/7000/linux/).
2. Add the following line to your notes.ini file

Extmgr\_addins=libkspam.so

If the line extmgr\_addins already appears in your notes.ini file append a comma and then the word "spam" to the end of the line, e.g.

Extmgr\_addins=av\_program,libkspam.so

The executables (kspam and libkspam.so) should be executable by the user who runs Domino.

### **Hereafter for both systems:**

3. Sign the database templates, kSpamCon.ntf, mailgood.ntf and mailspam.ntf.
4. Create the configuration database kSpamCon.nsf from the template kSpamCon.ntf.
5. Create the spam mail database mailspam.nsf from the template mailspam.ntf.
6. Create the mail database mailgood.nsf from the template mailgood.ntf.
7. Configure kSpam.

# kSpam

*– An open source Lotus Domino spam filter.*

## **4. Standard configuration**

kSpam uses extension manager addins to work, it should work even with a virus scanner that also uses extension manager addins.

To configure kSpam for standard operation, create a configuration document in the configuration database and complete the options on the 'General' tab. Then add some static rules to the rule list.

kSpamCon.nsf is the main configuration database which holds the rule list. You can also use some notes.ini environment variables to get kSpam to behave differently from its default settings.

mailgod.nsf is where copied mail is copied to. You can change the design of this database without problems or indeed you can replace it, mailgod.nsf is just an example, you can change the file name of the database kSpam copies to with the KS\_COPIED\_DB notes.ini variable.

When configured start the Domino server, kSpam loads it's rule list when incoming or outgoing mail starts to hit the server. You can also tell kSpam to reload the rule list every 60 minutes (or thereabouts) by adding KS\_RELOAD=1 to notes.ini (Version 1.1 or above).

A few example rules are listed in appendix A.

Once you have completed this stage you can start your Lotus Domino server and kSpam will operate on all your incoming emails.

# kSpam

*– An open source Lotus Domino spam filter.*

## 5. Bayesian setup and configuration

First please read a quick guide to Bayesian filtering theory at <http://www.paulgraham.com/spam.html>.

To configure kSpam for Bayesian operation, amend your kSpam configuration document in the configuration database, making sure you turn on the Bayesian filter.

For Bayesian filtering to work you need to have two sets of email for the filter to calculate the probability of tokens being found in good email, commonly called ham, and spam email.

The two databases that kSpam uses are called mailspam.nsf and mailgood.nsf, these two databases must be populated with the appropriate emails, I would recommend at least 1000 emails in each database before you turn on the Bayesian filter.

To help you collect emails for the Bayesian filter kSpam has two preparation phases you can use before you actually turn on Bayesian filtering. The two phases are named 'prep' and 'prep\_2', only one phase can be on at a time. When enabled the first phase (prep) makes kSpam copy one email for every ten that make it past the rule set to the database mailgood.nsf, this ratio can be changed in the configuration document. This first phase helps you populate the good mail database.

You must make sure only good emails stay in this database, and use the "Move selected messages to MailSpam" for any spam email that is caught in mailgood.nsf.

The second phase (prep\_2) will copy and emails with a probability of greater than 90% to mailspam.nsf and any emails with less than 10% probability to the mailgood.nsf database, this phase will only work if there are already spam emails in mailspam.nsf and good emails in mailgood.nsf.

Once both databases contain a reasonable amount of email, you can load the Bayesian probability addin and turn off the Bayesian preparation phases in the configuration document (make sure you leave the Bayesian filter turned on). Add "load" to the end of the 'Srvtasks' line in your notes.ini file, this will load the addin every time Domino starts. The first time the addin is started it may show an error about a file called comblist.txt, this is normal and it should not show the next time the addin is started.

If you have enough emails in mailspam.nsf and mailgood.nsf and have made the appropriate configuration changes in the configuration document you can now restart the server to turn on the Bayesian filtering.

# kSpam

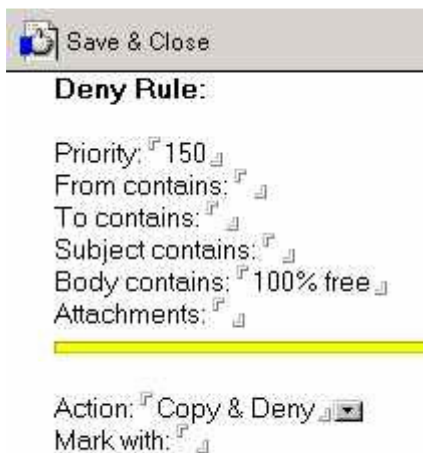
– *An open source Lotus Domino spam filter.*

## 6. Rules

In the main view of kSpamCon.nsf you can create an allow or deny rule. The rule list is read from top to bottom by kSpam so the rules with the lowest priority number are looked at first, as soon as an exact match is found the action for that rule is taken. If no rule matches the email the 'from' field is processed on it's own according to the notes.ini variables KS\_FILTER\_FROM\_INT, KS\_MAX\_FROM\_INTS, and KS\_MIN\_FROM\_LENGTH, then, if you have configured it, the Bayesian filtering is invoked on the email.

kSpam uses the rule list to do a case insensitive substring search of different email items. Currently you can not specify wildcards in a rule, so '\*.vbs' will not match any attachment with a .vbs extension but '.vbs' will, but you can now utilize ( as of version 1.37 beta ) PCRE regular expression matching (regex)\*, see pcrepattern.html for more information on specifying regular expressions (NB: regular expressions in the rule database must be prefixed with #REGEX#:).

### Rule Example:



Save & Close

**Deny Rule:**

Priority: 150

From contains:

To contains:

Subject contains:

Body contains: 100% free

Attachments:

---

Action: Copy & Deny

Mark with:

In this example the priority of 150 would mean that this rule gets run on an email before a rule with a priority of 200 but after a rule with a priority of 100.

The only other fields to contain anything are the Body contains field and Action field.

An email will only have the action specified by this rule run on it if the body of the email contains the phrase 100% free (ignoring case).

More rule examples in appendix A

# kSpam

*– An open source Lotus Domino spam filter.*

## **7. Limitations**

- kSpam has been tested on Lotus Domino R5.08 - R5.0.11, R6, R7.0 and R7.0.1
- Windows NT4, Windows 2000, Windows 2003
- Linux
- kSpam only filters incoming SMTP, NOT outgoing email.

# kSpam

– *An open source Lotus Domino spam filter.*

## 8. FAQs

How do I know if kSpam is running?	By sending a test internet message (inbound) through it with a subject/body containing a filtered word.
Does kSpam work on Linux?	Yes
Does kSpam work on a single server which acts as the SMTP gateway and main mail server for our organisation?	Yes.
Why do some of the default rules have words with spaces after/before them?	This is so kSpam will look for the word on it's own, i.e. ' porn ' matches the word 'porn' on it's own but not when it's part of a word.
Can I change the design of kSpamCon.nsf?	No, you'll break it.
Can I use kSpamCon.nsf via the web?	Yes.
What versions of Domino does kSpam work with?	I've tested it with 5.0.8 through 5.0.11, it also works with R6 and R7
How can I send you feedback/feature requests/comments?	By posting responses on openNTF.org or by emailing <kspam <at-here> tomlyne <dot> com>
Does kSpam work on Windows 2000?	Yes, it's been tested with: 5.0.11 and R6.0.3 on Windows 2000 Server and Windows 2000 Professional R7 on Windows 2003
Does kSpam implement a Bayesian filter?	It does from version 1.33 and above.
Why do I get lots of 'Note item not found' messages after running blood?	These seem to be generated by the CAPI. They are fixed in Version 1.34.
Does kSpam do regular expression (regex) matching?	Yes, as of version 1.37 beta
How do I stop the "MIME to CD Conversion..." messages when kSpam.blood runs?	Add the notes.ini variable CONVERTER_LOG_LEVEL=10
I like kSpam, can I donate money to the developers?	Yes, you can do it through OpenNTF.org from the kSpam project page.
Where can I get the latest version of kSpam?	From the kSpam project page at OpenNTF.org
Where can I find the kSpam source code?	There is anonymous CVS information available at <a href="https://sourceforge.net/cvs/?group_id=100598">https://sourceforge.net/cvs/?group_id=100598</a>

# kSpam

– *An open source Lotus Domino spam filter.*

## Appendix A – Rule examples

Rules are matches against in order of ascending rule priority. Regular expressions must be prefixed with '#REGEX#:'.

This rule will match any emails which contain '@spamdomain.com' in either the from header field or the mail from envelope field. If an email matches this criteria the email will be denied (i.e. it will be deleted).

Priority: (Required)	12
From contains: (optional)	@spamdomain.com
To contains: (optional)	
Subject contains: (optional)	
Body contains: (optional)	
Attachments: (optional)	
Action: (required)	Deny
Mark with: (optional, default is [?SPAM?])	

This rule will match any emails with '% off' in the subject. This includes any subjects like '10% off dvds', '5% off CDs' and 'Up to 50% off laptops'. If an email matches this criteria a copy of it will be stored in the copied mail database (default: mailspam.nsf) and it will then be denied.

Priority: (Required)	15
From contains: (optional)	
To contains: (optional)	
Subject contains: (optional)	% off
Body contains: (optional)	
Attachments: (optional)	
Action: (required)	copied & denied
Mark with: (optional, default is [?SPAM?])	

This rule will match any emails which have an attachment with '.vbs' in the filename. This includes filenames like 'myvirus.vbs', 'another\_virus.vbs.pif' etc.. If an email matches this criteria the subject will be changed to read '[!!!VIRUS!!!] <original subject>'.

Priority: (Required)	20
From contains: (optional)	

# kSpam

– *An open source Lotus Domino spam filter.*

To contains: (optional)	
Subject contains: (optional)	
Body contains: (optional)	
Attachments: (optional)	.vbs
Action: (required)	Mark
Mark with: (optional, default is [?SPAM?])	[!!VIRUS!!]

This rule would match any emails which have '@maybеспam.com' in the from: header field or mail from: envelope field and have 'hardcore' in the body. If the email matches these criteria it will be denied.

Priority: (Required)	30
From contains: (optional)	@maybеспam.com
To contains: (optional)	
Subject contains: (optional)	
Body contains: (optional)	Hardcore
Attachments: (optional)	
Action: (required)	Deny
Mark with: (optional, default is [?SPAM?])	

This rule would match any emails with 'porn ' in the subject. Take note of the space after the word 'porn'. This includes any subjects like 'hardcore porn you've never seen before' and 'teenyporn you will like' but NOT 'please lookout for pornography in your inbox'. If an email matches these criteria it will be denied.

Priority: (Required)	30
From contains: (optional)	
To contains: (optional)	
Subject contains: (optional)	porn
Body contains: (optional)	
Attachments: (optional)	
Action: (required)	Deny
Mark with: (optional, default is [?SPAM?])	

This rule would match any email with the PHRASE ' this is hardcore!! ' in the body. Take note of the spaces before and after the phrase. This would match any body which contained ' this is hardcore!! ' but not a body which started with 'this is hardcore!!' (because of the first space). If an email matches these criteria it will be denied.

# kSpam

– *An open source Lotus Domino spam filter.*

Priority: (Required)	100
From contains: (optional)	
To contains: (optional)	
Subject contains: (optional)	
Body contains: (optional)	this is hardcore!!
Attachments: (optional)	
Action: (required)	Deny
Mark with: (optional, default is [?SPAM?])	

Regular expressions:

This rule is a regular expression (NB: regular expressions must be prefixed with #REGEX#:) If the body of the message contains penis, penIs or pen1s, then the message will be denied.

Priority: (Required)	7
From contains: (optional)	
To contains: (optional)	
Subject contains: (optional)	
Body contains: (optional)	#REGEX#:pen[(?i)i 1]s
Attachments: (optional)	
Action: (required)	Deny
Mark with: (optional, default is [?SPAM?])	

## Appendix B - Notes.ini variables

As of version 1.4 all of these settings can be set in the configuration document in the kSpamCon.nsf database.

Use 'set conf <variable-name>=<value>' at the console or add to notes.ini.

variable-name	Example	Default	Description	Version
KS_BAYESIAN_ACTION	KS_BAYESIAN_ACTION=1	3	Define what to do with emails that have a probability above the Bayesian boundary (either the default of 90% or set by the user with KS_BAYESIAN_BOUNDARY). Can be 0 (Accept), 1 (Mark), 2 (Copy), 3 (Copy & Deny) or 4 (Deny). Notes: If you set this to 1 you should also set KS_BAYESIAN_ACTION_MARK_WITH.	1.4 and above.
KS_BAYESIAN_ACTION_MARK_WITH	KS_BAYESIAN_ACTION_MARK_WITH=[*This could be spam*]		Define what to mark the subject when KS_BAYESIAN_ACTION is set to 1.	1.4 and above.
KS_BAYESIAN_BOUNDARY	KS_BAYESIAN_BOUNDARY=95	90 (90%)	Set the probability limit for spam emails. Must be between 1 and 100, anything below 50 is a silly idea. Notes: Don't change this unless you know what you are doing.	1.37 beta and above.
KS_BAYESIAN_FILTER	KS_BAYESIAN_FILTER=1		Enable 'Bayesian' filtering. Copy all email with a probability of 90% or more to mailspam.nsf. Copy email that make it passed (the rule set and the Bayesian filter) one in KS_BAYESIAN_RATIO times. Notes: This must be set to 1 for any KS_BAYESIAN_XXXX variable to work.	1.33 and above.
KS_BAYESIAN_MARK	KS_BAYESIAN_MARK=1		Mark email that is scanned by the Bayesian filter with the probability (item: KS_BL_PROB) and the most interesting tokens (item: KS_BL_TOKENS). Use with: KS_BAYESIAN_FILTER	1.33 and above.

# kSpam

– *An open source Lotus Domino spam filter.*

KS_BAYESIAN_PREP	KS_BAYESIAN_PREP=1		Change target database of copied and denied messages to mailspam.nsf, copy one in KS_BAYESIAN_RATIO messages that make it past the rule set to mailgood.nsf. Use with: KS_BAYESIAN_FILTER Exclusive of: KS_BAYESIAN_PREP_2	1.33 and above.
KS_BAYESIAN_PREP_2	KS_BAYESIAN_PREP_2=1		Copy any incoming message with a probability above 90% to mailspam.nsf and any with a probability of less than 10% to mailgood.nsf. Use after KS_BAYESIAN_PREP. Use with: KS_BAYESIAN_FILTER Exclusive of: KS_BAYESIAN_PREP	1.33 and above.
KS_BAYESIAN_RATIO	KS_BAYESIAN_RATIO=20	10	Use with KS_BAYESIAN_FILTER and KS_BAYESIAN_PREP.	1.33 and above.
KS_BL_ADD			Additional probability added to the calculated probability to get the value in KS_BL_PROB, this is added when an email matches a rule with the action set to increase probability. Please note that rule matching does not stop when an email matches a rule with the action set to increase probability.	
KS_BL_DUMPLISTS	KS_BL_DUMPLISTS=1		Tell kSpam.blood to dump out probabilities and token counts to comblst.txt (final probabilities), goodlist.txt (good token count) and spamlist.txt (spam token count).	1.33 and above.
KS_BL_IGNORE	KS_BL_IGNORE=token1, token2, token3,		Tell kSpam.blood to ignore the specified tokens. Notes: This line must end with a comma ','.	1.35 and above
KS_BL_PERIOD	KS_BL_PERIOD=360		Set time in minutes between kspam.blood calculating probabilities for	1.33 and above.

# kSpam

– *An open source Lotus Domino spam filter.*

			Bayesian filtering. Notes: Calculating probabilities is resource intensive. Any value for this parameter under 180 is probably unnecessary.	
KS_BL_PROB			probability of the email being spam as calculated by the Bayesian filter.	
KS_BL_TOKENS			The probability of the 15 most interesting tokens in the email.	
KS_COPIED_DB	KS_COPIED_DB=junk.nsf		Tell kSpam to copy messages to this database. Must be a string and the database must exist.	1.21 and above.
KS_DEFAULT_PROB_INC	KS_DEFAULT_PROB_INC=10	0	Specify the default probability increase used when a value has not been specified by a rule with this action.	1.4b and above
KS_DEFAULTACTION	KS_DEFAULTACTION=3		Can be 0 (Accept), 1 (Mark), 2 (Copy), 3 (Copy & Deny), 4 (Deny) or 5 (Increase Probability). Defines what to do with the messages matched by KS_MIN_FROM_LENGTH, KS_MAX_FROM_INTS and KS_FILTER_FROM_INT.	1.22 and above.
KS_FILTER_FROM_INT	KS_FILTER_FROM_INT=1		If set to 1 then the first character of the senders username cannot be an integer. Must be 1 or 0.	1.22 and above.
KS_IID	KS_IIS=ks1234		Set instance ID for servers in organisation, this stops kSpam scanning an email more than once as it passes through multiple installations of kSpam. Notes: Only first 6 characters processed. If this is not present kSpam looks at the first 6 characters of the servers Domain item in it's server document. This will also stop email sent on from the copied.nsf (or mailspam.nsf) database	1.39b and above

# kSpam

– *An open source Lotus Domino spam filter.*

			being caught again.	
KS_INTERESTING_FORMS	OtherForm,AndAnotherForm,		Define other forms for kSpam to scan, Memo and Reply are included by default. Notes: The forms specified must be delimited by and end with a comma.	1.4 and above.
KS_MARK	KS_MARK=1		Tells kSpam to mark any copied & denied message with a KS_REASON notes item.	1.21 and above.
KS_MAX_FROM_INTS	KS_MAX_FROM_INTS=3		Defines the maximum number of single integers in the senders username (Before @).	1.22 and above.
KS_MIN_FROM_LENGTH	KS_MIN_FROM_LENGTH=1		Defines minimum string length (strlen()) of the From field.	1.22 and above.
KS_REASON			The reason why the email had an action carried out on it.	
KS_RECIPIENTS	KS_RECIPIENTS=1		Add a KS_RECIPIENTS item to copied and denied emails. This item is set as the readers field of the Notes document in the database so only the recipients of the original email and anyone in the group 'kSpamAdministrators' will be able to see the document. Notes: The username portion of a user's email address, i.e. the part before the '@' sign, must be present in the 'User name' field of their person document.	1.4 and above.
KS_RELOAD	KS_RELOAD=1		Tells kSpam to reload rule list every 60 minutes (or thereabouts).	1.1 and above.
KS_STATS	KS_STATS=1		Tells kSpam to log stats (show by console command 'show stat smtp.kspam.*'). See 'By Rule Number' in kSpamConf.nsf to help with the above notes.ini variable. Don't use with: KS_RELOAD (You may get mixed up with the rule numbers if you add extra rules or delete rules	1.32 and above.

## Appendix C – Regular expression references

Regular expression support is provided by the PCRE library package, which is open source software, written by Philip Hazel, and copyright by the University of Cambridge, England. Source:

<ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/>

- \* [PCRE REGULAR EXPRESSION DETAILS](#)
- \* [BACKSLASH](#)
- \* [CIRCUMFLEX AND DOLLAR](#)
- \* [FULL STOP \(PERIOD, DOT\)](#)
- \* [MATCHING A SINGLE BYTE](#)
- \* [SQUARE BRACKETS](#)
- \* [POSIX CHARACTER CLASSES](#)
- \* [VERTICAL BAR](#)
- \* [INTERNAL OPTION SETTING](#)
- \* [SUBPATTERNS](#)
- \* [NAMED SUBPATTERNS](#)
- \* [REPETITION](#)
- \* [ATOMIC GROUPING AND POSSESSIVE QUANTIFIERS](#)
- \* [BACK REFERENCES](#)
- \* [ASSERTIONS](#)
- \* [CONDITIONAL SUBPATTERNS](#)
- \* [COMMENTS](#)
- \* [RECURSIVE PATTERNS](#)
- \* [SUBPATTERNS AS SUBROUTINES](#)
- \* [CALLOUTS](#)

### PCRE REGULAR EXPRESSION DETAILS

The syntax and semantics of the regular expressions supported by PCRE are described below. Regular expressions are also described in the Perl documentation and in a number of other books, some of which have copious examples. Jeffrey Friedl's "Mastering Regular Expressions", published by O'Reilly, covers them in great detail. The description here is intended as reference documentation.

The basic operation of PCRE is on strings of bytes. However, there is also support for UTF-8 character strings. To use this support you must build PCRE to include UTF-8 support, and then call `pcre_compile()` with the `PCRE_UTF8` option. How this affects the pattern matching is mentioned in several places below. There is also a summary of UTF-8 features in the section on UTF-8 support in the main `pcre` page.

# kSpam

– *An open source Lotus Domino spam filter.*

A regular expression is a pattern that is matched against a subject string from left to right. Most characters stand for themselves in a pattern, and match the corresponding characters in the subject. As a trivial example, the pattern

The quick brown fox

matches a portion of a subject string that is identical to itself. The power of regular expressions comes from the ability to include alternatives and repetitions in the pattern. These are encoded in the pattern by the use of meta-characters, which do not stand for themselves but instead are interpreted in some special way.

There are two different sets of meta-characters: those that are recognized anywhere in the pattern except within square brackets, and those that are recognized in square brackets. Outside square brackets, the meta-characters are as follows:

- \ general escape character with several uses
- ^ assert start of string (or line, in multiline mode)
- \$ assert end of string (or line, in multiline mode)
- . match any character except newline (by default)
- [ start character class definition
- | start of alternative branch
- ( start subpattern
- ) end subpattern
- ? extends the meaning of (  
also 0 or 1 quantifier  
also quantifier minimizer
- \* 0 or more quantifier
- + 1 or more quantifier  
also "possessive quantifier"
- { start min/max quantifier

Part of a pattern that is in square brackets is called a "character class". In a character class the only meta-characters are:

- \ general escape character
- ^ negate the class, but only if the first character
- indicates character range
- [ POSIX character class (only if followed by POSIX syntax)
- ] terminates the character class

The following sections describe the use of each of the meta-characters.

## **BACKSLASH**

# kSpam

## – *An open source Lotus Domino spam filter.*

The backslash character has several uses. Firstly, if it is followed by a non-alphameric character, it takes away any special meaning that character may have. This use of backslash as an escape character applies both inside and outside character classes.

For example, if you want to match a `*` character, you write `\*` in the pattern. This escaping action applies whether or not the following character would otherwise be interpreted as a meta-character, so it is always safe to precede a non-alphameric with backslash to specify that it stands for itself. In particular, if you want to match a backslash, you write `\\`.

If a pattern is compiled with the `PCRE_EXTENDED` option, whitespace in the pattern (other than in a character class) and characters between a `#` outside a character class and the next newline character are ignored. An escaping backslash can be used to include a whitespace or `#` character as part of the pattern.

If you want to remove the special meaning from a sequence of characters, you can do so by putting them between `\Q` and `\E`. This is different from Perl in that `$` and `@` are handled as literals in `\Q...\E` sequences in PCRE, whereas in Perl, `$` and `@` cause variable interpolation. Note the following examples:

Pattern	PCRE matches	Perl matches
<code>\Qabc\$xyz\E</code>	<code>abc\$xyz</code>	<code>abc</code> followed by the contents of <code>\$xyz</code>
<code>\Qabc\ \$xyz\E</code>	<code>abc\ \$xyz</code>	<code>abc\ \$xyz</code>
<code>\Qabc\E\ \$Qxyz\E</code>	<code>abc\$xyz</code>	<code>abc\$xyz</code>

The `\Q...\E` sequence is recognized both inside and outside character classes.

A second use of backslash provides a way of encoding non-printing characters in patterns in a visible manner. There is no restriction on the appearance of non-printing characters, apart from the binary zero that terminates a pattern, but when a pattern is being prepared by text editing, it is usually easier to use one of the following escape sequences than the binary character it represents:

<code>\a</code>	alarm, that is, the BEL character (hex 07)
<code>\cx</code>	"control-x", where x is any character
<code>\e</code>	escape (hex 1B)
<code>\f</code>	formfeed (hex 0C)
<code>\n</code>	newline (hex 0A)
<code>\r</code>	carriage return (hex 0D)
<code>\t</code>	tab (hex 09)
<code>\ddd</code>	character with octal code ddd, or backreference
<code>\xhh</code>	character with hex code hh
<code>\x{hhh..}</code>	character with hex code hhh... (UTF-8 mode only)

# kSpam

## *– An open source Lotus Domino spam filter.*

The precise effect of `\cx` is as follows: if `x` is a lower case letter, it is converted to upper case. Then bit 6 of the character (hex 40) is inverted. Thus `\cz` becomes hex 1A, but `\c{` becomes hex 3B, while `\c;` becomes hex 7B.

After `\x`, from zero to two hexadecimal digits are read (letters can be in upper or lower case). In UTF-8 mode, any number of hexadecimal digits may appear between `\x{` and `}`, but the value of the character code must be less than  $2^{31}$  (that is, the maximum hexadecimal value is 7FFFFFFF). If characters other than hexadecimal digits appear between `\x{` and `}`, or if there is no terminating `}`, this form of escape is not recognized. Instead, the initial `\x` will be interpreted as a basic hexadecimal escape, with no following digits, giving a byte whose value is zero.

Characters whose value is less than 256 can be defined by either of the two syntaxes for `\x` when PCRE is in UTF-8 mode. There is no difference in the way they are handled. For example, `\xdc` is exactly the same as `\x{dc}`.

After `\0` up to two further octal digits are read. In both cases, if there are fewer than two digits, just those that are present are used. Thus the sequence `\0\x\07` specifies two binary zeros followed by a BEL character (code value 7). Make sure you supply two digits after the initial zero if the character that follows is itself an octal digit.

The handling of a backslash followed by a digit other than 0 is complicated. Outside a character class, PCRE reads it and any following digits as a decimal number. If the number is less than 10, or if there have been at least that many previous capturing left parentheses in the expression, the entire sequence is taken as a back reference. A description of how this works is given later, following the discussion of parenthesized subpatterns.

Inside a character class, or if the decimal number is greater than 9 and there have not been that many capturing subpatterns, PCRE re-reads up to three octal digits following the backslash, and generates a single byte from the least significant 8 bits of the value. Any subsequent digits stand for themselves. For example:

- `\040` is another way of writing a space
- `\40` is the same, provided there are fewer than 40 previous capturing subpatterns
- `\7` is always a back reference
- `\11` might be a back reference, or another way of writing a tab
- `\011` is always a tab
- `\0113` is a tab followed by the character "3"
- `\113` might be a back reference, otherwise the character with octal code 113
- `\377` might be a back reference, otherwise the byte consisting entirely of 1 bits

# kSpam

– *An open source Lotus Domino spam filter.*

`\81` is either a back reference, or a binary zero followed by the two characters "8" and "1"

Note that octal values of 100 or greater must not be introduced by a leading zero, because no more than three octal digits are ever read.

All the sequences that define a single byte value or a single UTF-8 character (in UTF-8 mode) can be used both inside and outside character classes. In addition, inside a character class, the sequence `\b` is interpreted as the backspace character (hex 08). Outside a character class it has a different meaning (see below).

The third use of backslash is for specifying generic character types:

- `\d` any decimal digit
- `\D` any character that is not a decimal digit
- `\s` any whitespace character
- `\S` any character that is not a whitespace character
- `\w` any "word" character
- `\W` any "non-word" character

Each pair of escape sequences partitions the complete set of characters into two disjoint sets. Any given character matches one, and only one, of each pair.

In UTF-8 mode, characters with values greater than 255 never match `\d`, `\s`, or `\w`, and always match `\D`, `\S`, and `\W`.

For compatibility with Perl, `\s` does not match the VT character (code 11). This makes it different from the the POSIX "space" class. The `\s` characters are HT (9), LF (10), FF (12), CR (13), and space (32).

A "word" character is any letter or digit or the underscore character, that is, any character which can be part of a Perl "word". The definition of letters and digits is controlled by PCRE's character tables, and may vary if locale-specific matching is taking place (see "Locale support" in the `pcreapi` page). For example, in the "fr" (French) locale, some character codes greater than 128 are used for accented letters, and these are matched by `\w`.

These character type sequences can appear both inside and outside character classes. They each match one character of the appropriate type. If the current matching point is at the end of the subject string, all of them fail, since there is no character to match.

The fourth use of backslash is for certain simple assertions. An assertion specifies a condition that has to be met at a particular point in a match, without consuming any characters from the subject string. The use of subpatterns for more complicated assertions is described below. The backslashed assertions are

# kSpam

– *An open source Lotus Domino spam filter.*

- `\b` matches at a word boundary
- `\B` matches when not at a word boundary
- `\A` matches at start of subject
- `\Z` matches at end of subject or before newline at end
- `\z` matches at end of subject
- `\G` matches at first matching position in subject

These assertions may not appear in character classes (but note that `\b` has a different meaning, namely the backspace character, inside a character class).

A word boundary is a position in the subject string where the current character and the previous character do not both match `\w` or `\W` (i.e. one matches `\w` and the other matches `\W`), or the start or end of the string if the first or last character matches `\w`, respectively.

The `\A`, `\Z`, and `\z` assertions differ from the traditional circumflex and dollar (described below) in that they only ever match at the very start and end of the subject string, whatever options are set. Thus, they are independent of multiline mode.

They are not affected by the `PCRE_NOTBOL` or `PCRE_NOTEOL` options. If the `startoffset` argument of `pcre_exec()` is non-zero, indicating that matching is to start at a point other than the beginning of the subject, `\A` can never match. The difference between `\Z` and `\z` is that `\Z` matches before a newline that is the last character of the string as well as at the end of the string, whereas `\z` matches only at the end.

The `\G` assertion is true only when the current matching position is at the start point of the match, as specified by the `startoffset` argument of `pcre_exec()`. It differs from `\A` when the value of `startoffset` is non-zero. By calling `pcre_exec()` multiple times with appropriate arguments, you can mimic Perl's `/g` option, and it is in this kind of implementation where `\G` can be useful.

Note, however, that PCRE's interpretation of `\G`, as the start of the current match, is subtly different from Perl's, which defines it as the end of the previous match. In Perl, these can be different when the previously matched string was empty. Because PCRE does just one match at a time, it cannot reproduce this behaviour.

If all the alternatives of a pattern begin with `\G`, the expression is anchored to the starting match position, and the "anchored" flag is set in the compiled regular expression.

## CIRCUMFLEX AND DOLLAR

# kSpam

– *An open source Lotus Domino spam filter.*

Outside a character class, in the default matching mode, the circumflex character is an assertion which is true only if the current matching point is at the start of the subject string. If the `startoffset` argument of `pcre_exec()` is non-zero, circumflex can never match if the `PCRE_MULTILINE` option is unset. Inside a character class, circumflex has an entirely different meaning (see below).

Circumflex need not be the first character of the pattern if a number of alternatives are involved, but it should be the first thing in each alternative in which it appears if the pattern is ever to match that branch. If all possible alternatives start with a circumflex, that is, if the pattern is constrained to match only at the start of the subject, it is said to be an "anchored" pattern. (There are also other constructs that can cause a pattern to be anchored.)

A dollar character is an assertion which is true only if the current matching point is at the end of the subject string, or immediately before a newline character that is the last character in the string (by default). Dollar need not be the last character of the pattern if a number of alternatives are involved, but it should be the last item in any branch in which it appears. Dollar has no special meaning in a character class.

The meaning of dollar can be changed so that it matches only at the very end of the string, by setting the `PCRE_DOLLAR_ENDONLY` option at compile time. This does not affect the `\Z` assertion.

The meanings of the circumflex and dollar characters are changed if the `PCRE_MULTILINE` option is set. When this is the case, they match immediately after and immediately before an internal newline character, respectively, in addition to matching at the start and end of the subject string. For example, the pattern `/^abc$/` matches the subject string "def\nabc" in multiline mode, but not otherwise. Consequently, patterns that are anchored in single line mode because all branches start with `^` are not anchored in multiline mode, and a match for circumflex is possible when the `startoffset` argument of `pcre_exec()` is non-zero. The `PCRE_DOLLAR_ENDONLY` option is ignored if `PCRE_MULTILINE` is set.

Note that the sequences `\A`, `\Z`, and `\z` can be used to match the start and end of the subject in both modes, and if all branches of a pattern start with `\A` it is always anchored, whether `PCRE_MULTILINE` is set or not.

## **FULL STOP (PERIOD, DOT)**

Outside a character class, a dot in the pattern matches any one character in the subject, including a non-printing character, but not (by default) newline. In UTF-8 mode, a dot matches any UTF-8 character, which might be more than one byte long, except (by default) for newline. If the `PCRE_DOTALL` option is set, dots match newlines as well. The handling of dot is entirely independent of

# kSpam

– *An open source Lotus Domino spam filter.*

the handling of circumflex and dollar, the only relationship being that they both involve newline characters. Dot has no special meaning in a character class.

## **MATCHING A SINGLE BYTE**

Outside a character class, the escape sequence `\C` matches any one byte, both in and out of UTF-8 mode. Unlike a dot, it always matches a newline. The feature is provided in Perl in order to match individual bytes in UTF-8 mode. Because it breaks up UTF-8 characters into individual bytes, what remains in the string may be a malformed UTF-8 string. For this reason it is best avoided.

PCRE does not allow `\C` to appear in lookbehind assertions (see below), because in UTF-8 mode it makes it impossible to calculate the length of the lookbehind.

## **SQUARE BRACKETS**

An opening square bracket introduces a character class, terminated by a closing square bracket. A closing square bracket on its own is not special. If a closing square bracket is required as a member of the class, it should be the first data character in the class (after an initial circumflex, if present) or escaped with a backslash.

A character class matches a single character in the subject. In UTF-8 mode, the character may occupy more than one byte. A matched character must be in the set of characters defined by the class, unless the first character in the class definition is a circumflex, in which case the subject character must not be in the set defined by the class. If a circumflex is actually required as a member of the class, ensure it is not the first character, or escape it with a backslash.

For example, the character class `[aeiou]` matches any lower case vowel, while `[^aeiou]` matches any character that is not a lower case vowel. Note that a circumflex is just a convenient notation for specifying the characters which are in the class by enumerating those that are not. It is not an assertion: it still consumes a character from the subject string, and fails if the current pointer is at the end of the string.

In UTF-8 mode, characters with values greater than 255 can be included in a class as a literal string of bytes, or by using the `\x{}` escaping mechanism.

When caseless matching is set, any letters in a class represent both their upper case and lower case versions, so for example, a caseless `[aeiou]` matches "A" as well as "a", and a caseless `[^aeiou]` does not match "A", whereas a careful version would. PCRE does not support the concept of case for characters with values greater than 255.

# kSpam

## *– An open source Lotus Domino spam filter.*

The newline character is never treated in any special way in character classes, whatever the setting of the PCRE\_DOTALL or PCRE\_MULTILINE options is. A class such as `[^a]` will always match a newline.

The minus (hyphen) character can be used to specify a range of characters in a character class. For example, `[d-m]` matches any letter between d and m, inclusive. If a minus character is required in a class, it must be escaped with a backslash or appear in a position where it cannot be interpreted as indicating a range, typically as the first or last character in the class.

It is not possible to have the literal character "]" as the end character of a range. A pattern such as `[W-]46]` is interpreted as a class of two characters ("W" and "-") followed by a literal string "46]", so it would match "W46]" or "-46]". However, if the "]" is escaped with a backslash it is interpreted as the end of range, so `[W-\]46]` is interpreted as a single class containing a range followed by two separate characters. The octal or hexadecimal representation of "]" can also be used to end a range.

Ranges operate in the collating sequence of character values. They can also be used for characters specified numerically, for example `[\000-\037]`. In UTF-8 mode, ranges can include characters whose values are greater than 255, for example `[\x{100}-\x{2ff}]`.

If a range that includes letters is used when caseless matching is set, it matches the letters in either case. For example, `[W-c]` is equivalent to `[][\^_`wxyzabc]`, matched caselessly, and if character tables for the "fr" locale are in use, `[\xc8-\xcb]` matches accented E characters in both cases.

The character types `\d`, `\D`, `\s`, `\S`, `\w`, and `\W` may also appear in a character class, and add the characters that they match to the class. For example, `[\dABCDEF]` matches any hexadecimal digit. A circumflex can conveniently be used with the upper case character types to specify a more restricted set of characters than the matching lower case type. For example, the class `[\W_]` matches any letter or digit, but not underscore.

All non-alphanumeric characters other than `\`, `-`, `^` (at the start) and the terminating `]` are non-special in character classes, but it does no harm if they are escaped.

## **POSIX CHARACTER CLASSES**

Perl supports the POSIX notation for character classes, which uses names enclosed by `[:` and `:]` within the enclosing square brackets. PCRE also supports this notation. For example,

```
[01[:alpha:]]%
```

# kSpam

– *An open source Lotus Domino spam filter.*

matches "0", "1", any alphabetic character, or "%". The supported class names are

alnum	letters and digits
alpha	letters
ascii	character codes 0 - 127
blank	space or tab only
cntrl	control characters
digit	decimal digits (same as \d)
graph	printing characters, excluding space
lower	lower case letters
print	printing characters, including space
punct	printing characters, excluding letters and digits
space	white space (not quite the same as \s)
upper	upper case letters
word	"word" characters (same as \w)
xdigit	hexadecimal digits

The "space" characters are HT (9), LF (10), VT (11), FF (12), CR (13), and space (32). Notice that this list includes the VT character (code 11). This makes "space" different to \s, which does not include VT (for Perl compatibility).

The name "word" is a Perl extension, and "blank" is a GNU extension from Perl 5.8. Another Perl extension is negation, which is indicated by a ^ character after the colon. For example,

```
[12[:^digit:]]
```

matches "1", "2", or any non-digit. PCRE (and Perl) also recognize the POSIX syntax [.ch.] and [=ch=] where "ch" is a "collating element", but these are not supported, and an error is given if they are encountered.

In UTF-8 mode, characters with values greater than 255 do not match any of the POSIX character classes.

## VERTICAL BAR

Vertical bar characters are used to separate alternative patterns. For example, the pattern

```
gilbert|sullivan
```

matches either "gilbert" or "sullivan". Any number of alternatives may appear, and an empty alternative is permitted (matching the empty string). The matching process tries each alternative in turn, from left to right, and the first one that succeeds is used. If the alternatives are within a subpattern (defined

# kSpam

– *An open source Lotus Domino spam filter.*

below), "succeeds" means matching the rest of the main pattern as well as the alternative in the subpattern.

## INTERNAL OPTION SETTING

The settings of the PCRE\_CASELESS, PCRE\_MULTILINE, PCRE\_DOTALL, and PCRE\_EXTENDED options can be changed from within the pattern by a sequence of Perl option letters enclosed between "(?" and ")". The option letters are

- i for PCRE\_CASELESS
- m for PCRE\_MULTILINE
- s for PCRE\_DOTALL
- x for PCRE\_EXTENDED

For example, (?im) sets caseless, multiline matching. It is also possible to unset these options by preceding the letter with a hyphen, and a combined setting and unsetting such as (?im-sx), which sets PCRE\_CASELESS and PCRE\_MULTILINE while unsetting PCRE\_DOTALL and PCRE\_EXTENDED, is also permitted. If a letter appears both before and after the hyphen, the option is unset.

When an option change occurs at top level (that is, not inside subpattern parentheses), the change applies to the remainder of the pattern that follows. If the change is placed right at the start of a pattern, PCRE extracts it into the global options (and it will therefore show up in data extracted by the pcre\_fullinfo() function).

An option change within a subpattern affects only that part of the current pattern that follows it, so

(a(?i)b)c

matches abc and aBc and no other strings (assuming PCRE\_CASELESS is not used). By this means, options can be made to have different settings in different parts of the pattern. Any changes made in one alternative do carry on into subsequent branches within the same subpattern. For example,

(a(?i)b|c)

matches "ab", "aB", "c", and "C", even though when matching "C" the first branch is abandoned before the option setting. This is because the effects of option settings happen at compile time. There would be some very weird behaviour otherwise.

The PCRE-specific options PCRE\_UNGREEDY and PCRE\_EXTRA can be changed in the same way as the Perl-compatible options by using the characters U and

# kSpam

– *An open source Lotus Domino spam filter.*

X respectively. The (?X) flag setting is special in that it must always occur earlier in the pattern than any of the additional features it turns on, even when it is at top level. It is best put at the start.

## SUBPATTERNS

Subpatterns are delimited by parentheses (round brackets), which can be nested. Marking part of a pattern as a subpattern does two things:

1. It localizes a set of alternatives. For example, the pattern

```
cat(aract|erpillar|)
```

matches one of the words "cat", "cataract", or "caterpillar". Without the parentheses, it would match "cataract", "erpillar" or the empty string.

2. It sets up the subpattern as a capturing subpattern (as defined above).

When the whole pattern matches, that portion of the subject string that matched the subpattern is passed back to the caller via the ovector argument of `pcre_exec()`. Opening parentheses are counted from left to right (starting from 1) to obtain the numbers of the capturing subpatterns.

For example, if the string "the red king" is matched against the pattern

```
the ((red|white) (king|queen))
```

the captured substrings are "red king", "red", and "king", and are numbered 1, 2, and 3, respectively.

The fact that plain parentheses fulfil two functions is not always helpful. There are often times when a grouping subpattern is required without a capturing requirement. If an opening parenthesis is followed by a question mark and a colon, the subpattern does not do any capturing, and is not counted when computing the number of any subsequent capturing subpatterns. For example, if the string "the white queen" is matched against the pattern

```
the ((?:red|white) (king|queen))
```

the captured substrings are "white queen" and "queen", and are numbered 1 and 2. The maximum number of capturing subpatterns is 65535, and the maximum depth of nesting of all subpatterns, both capturing and non-capturing, is 200.

As a convenient shorthand, if any option settings are required at the start of a non-capturing subpattern, the option letters may appear between the "?" and the ":". Thus the two patterns

# kSpam

– *An open source Lotus Domino spam filter.*

`(?i:saturday|sunday)`

`(?:(?i)saturday|sunday)`

match exactly the same set of strings. Because alternative branches are tried from left to right, and options are not reset until the end of the subpattern is reached, an option setting in one branch does affect subsequent branches, so the above patterns match "SUNDAY" as well as "Saturday".

## NAMED SUBPATTERNS

Identifying capturing parentheses by number is simple, but it can be very hard to keep track of the numbers in complicated regular expressions. Furthermore, if an expression is modified, the numbers may change. To help with the difficulty, PCRE supports the naming of subpatterns, something that Perl does not provide. The Python syntax `(?P<name>...)` is used. Names consist of alphanumeric characters and underscores, and must be unique within a pattern.

Named capturing parentheses are still allocated numbers as well as names. The PCRE API provides function calls for extracting the name-to-number translation table from a compiled pattern. For further details see the `pcreapi` documentation.

## REPETITION

Repetition is specified by quantifiers, which can follow any of the following items:

- a literal data character
- the `.` metacharacter
- the `\C` escape sequence
- escapes such as `\d` that match single characters
- a character class
- a back reference (see next section)
- a parenthesized subpattern (unless it is an assertion)

The general repetition quantifier specifies a minimum and maximum number of permitted matches, by giving the two numbers in curly brackets (braces), separated by a comma. The numbers must be less than 65536, and the first must be less than or equal to the second. For example:

`z{2,4}`

matches "zz", "zzz", or "zzzz". A closing brace on its own is not a special character. If the second number is omitted, but the comma is present, there is

# kSpam

– *An open source Lotus Domino spam filter.*

no upper limit; if the second number and the comma are both omitted, the quantifier specifies an exact number of required matches. Thus

```
[aeiou]{3,}
```

matches at least 3 successive vowels, but may match many more, while

```
\d{8}
```

matches exactly 8 digits. An opening curly bracket that appears in a position where a quantifier is not allowed, or one that does not match the syntax of a quantifier, is taken as a literal character. For example, `{,6}` is not a quantifier, but a literal string of four characters.

In UTF-8 mode, quantifiers apply to UTF-8 characters rather than to individual bytes. Thus, for example, `\x{100}{2}` matches two UTF-8 characters, each of which is represented by a two-byte sequence.

The quantifier `{0}` is permitted, causing the expression to behave as if the previous item and the quantifier were not present.

For convenience (and historical compatibility) the three most common quantifiers have single-character abbreviations:

- \* is equivalent to `{0,}`
- + is equivalent to `{1,}`
- ? is equivalent to `{0,1}`

It is possible to construct infinite loops by following a subpattern that can match no characters with a quantifier that has no upper limit, for example:

```
(a?)*
```

Earlier versions of Perl and PCRE used to give an error at compile time for such patterns. However, because there are cases where this can be useful, such patterns are now accepted, but if any repetition of the subpattern does in fact match no characters, the loop is forcibly broken.

By default, the quantifiers are "greedy", that is, they match as much as possible (up to the maximum number of permitted times), without causing the rest of the pattern to fail. The classic example of where this gives problems is in trying to match comments in C programs. These appear between the sequences `/*` and `*/` and within the sequence, individual `*` and `/` characters may appear. An attempt to match C comments by applying the pattern

```
/\*.*\*/
```

to the string

# kSpam

– *An open source Lotus Domino spam filter.*

```
/* first command */ not comment /* second comment */
```

fails, because it matches the entire string owing to the greediness of the `.*` item.

However, if a quantifier is followed by a question mark, it ceases to be greedy, and instead matches the minimum number of times possible, so the pattern

```
/\*.*?\*/
```

does the right thing with the C comments. The meaning of the various quantifiers is not otherwise changed, just the preferred number of matches. Do not confuse this use of question mark with its use as a quantifier in its own right. Because it has two uses, it can sometimes appear doubled, as in

```
\d??\d
```

which matches one digit by preference, but can match two if that is the only way the rest of the pattern matches.

If the `PCRE_UNGREEDY` option is set (an option which is not available in Perl), the quantifiers are not greedy by default, but individual ones can be made greedy by following them with a question mark. In other words, it inverts the default behaviour.

When a parenthesized subpattern is quantified with a minimum repeat count that is greater than 1 or with a limited maximum, more store is required for the compiled pattern, in proportion to the size of the minimum or maximum.

If a pattern starts with `.*` or `{0,}` and the `PCRE_DOTALL` option (equivalent to Perl's `/s`) is set, thus allowing the `.` to match newlines, the pattern is implicitly anchored, because whatever follows will be tried against every character position in the subject string, so there is no point in retrying the overall match at any position after the first. PCRE normally treats such a pattern as though it were preceded by `\A`.

In cases where it is known that the subject string contains no newlines, it is worth setting `PCRE_DOTALL` in order to obtain this optimization, or alternatively using `^` to indicate anchoring explicitly.

However, there is one situation where the optimization cannot be used. When `.*` is inside capturing parentheses that are the subject of a backreference elsewhere in the pattern, a match at the start may fail, and a later one succeed. Consider, for example:

```
(.*)abc\1
```

# kSpam

– *An open source Lotus Domino spam filter.*

If the subject is "xyz123abc123" the match point is the fourth character. For this reason, such a pattern is not implicitly anchored.

When a capturing subpattern is repeated, the value captured is the substring that matched the final iteration. For example, after

```
(tweedle[dume]{3}\s*)+
```

has matched "tweedledum tweedledee" the value of the captured substring is "tweedledee". However, if there are nested capturing subpatterns, the corresponding captured values may have been set in previous iterations. For example, after

```
/(a|(b))+/
```

matches "aba" the value of the second captured substring is "b".

## ATOMIC GROUPING AND POSSESSIVE QUANTIFIERS

With both maximizing and minimizing repetition, failure of what follows normally causes the repeated item to be re-evaluated to see if a different number of repeats allows the rest of the pattern to match. Sometimes it is useful to prevent this, either to change the nature of the match, or to cause it fail earlier than it otherwise might, when the author of the pattern knows there is no point in carrying on.

Consider, for example, the pattern `\d+foo` when applied to the subject line

```
123456bar
```

After matching all 6 digits and then failing to match "foo", the normal action of the matcher is to try again with only 5 digits matching the `\d+` item, and then with 4, and so on, before ultimately failing. "Atomic grouping" (a term taken from Jeffrey Friedl's book) provides the means for specifying that once a subpattern has matched, it is not to be re-evaluated in this way.

If we use atomic grouping for the previous example, the matcher would give up immediately on failing to match "foo" the first time. The notation is a kind of special parenthesis, starting with `(?>` as in this example:

```
(?>\d+)bar
```

This kind of parenthesis "locks up" the part of the pattern it contains once it has matched, and a failure further into the pattern is prevented from backtracking into it. Backtracking past it to previous items, however, works as normal.

# kSpam

– *An open source Lotus Domino spam filter.*

An alternative description is that a subpattern of this type matches the string of characters that an identical standalone pattern would match, if anchored at the current point in the subject string.

Atomic grouping subpatterns are not capturing subpatterns. Simple cases such as the above example can be thought of as a maximizing repeat that must swallow everything it can. So, while both `\d+` and `\d+?` are prepared to adjust the number of digits they match in order to make the rest of the pattern match, `(?>\d+)` can only match an entire sequence of digits.

Atomic groups in general can of course contain arbitrarily complicated subpatterns, and can be nested. However, when the subpattern for an atomic group is just a single repeated item, as in the example above, a simpler notation, called a "possessive quantifier" can be used. This consists of an additional `+` character following a quantifier. Using this notation, the previous example can be rewritten as

```
\d++bar
```

Possessive quantifiers are always greedy; the setting of the `PCRE_UNGREEDY` option is ignored. They are a convenient notation for the simpler forms of atomic group. However, there is no difference in the meaning or processing of a possessive quantifier and the equivalent atomic group.

The possessive quantifier syntax is an extension to the Perl syntax. It originates in Sun's Java package.

When a pattern contains an unlimited repeat inside a subpattern that can itself be repeated an unlimited number of times, the use of an atomic group is the only way to avoid some failing matches taking a very long time indeed. The pattern

```
(\D+|<\d+>)*[!?]
```

matches an unlimited number of substrings that either consist of non-digits, or digits enclosed in `<>`, followed by either `!` or `?`. When it matches, it runs quickly. However, if it is applied to

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

it takes a long time before reporting failure. This is because the string can be divided between the two repeats in a large number of ways, and all have to be tried. (The example used `[!?]` rather than a single character at the end, because both PCRE and Perl have an optimization that allows for fast failure when a single character is used. They remember the last single character that is required for a match, and fail early if it is not present in the string.) If the pattern is changed to

# kSpam

– *An open source Lotus Domino spam filter.*

```
((?>\D+)|<\d+>)*[!?
```

sequences of non-digits cannot be broken, and failure happens quickly.

## BACK REFERENCES

Outside a character class, a backslash followed by a digit greater than 0 (and possibly further digits) is a back reference to a capturing subpattern earlier (that is, to its left) in the pattern, provided there have been that many previous capturing left parentheses.

However, if the decimal number following the backslash is less than 10, it is always taken as a back reference, and causes an error only if there are not that many capturing left parentheses in the entire pattern. In other words, the parentheses that are referenced need not be to the left of the reference for numbers less than 10. See the section entitled "Backslash" above for further details of the handling of digits following a backslash.

A back reference matches whatever actually matched the capturing subpattern in the current subject string, rather than anything matching the subpattern itself (see "Subpatterns as subroutines" below for a way of doing that). So the pattern

```
(sens|respons)e and \1ibility
```

matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If careful matching is in force at the time of the back reference, the case of letters is relevant. For example,

```
((?i)rah)\s+\1
```

matches "rah rah" and "RAH RAH", but not "RAH rah", even though the original capturing subpattern is matched caselessly.

Back references to named subpatterns use the Python syntax (?P=name). We could rewrite the above example as follows:

```
(?<p1>(i)rah)\s+(?P=p1)
```

There may be more than one back reference to the same subpattern. If a subpattern has not actually been used in a particular match, any back references to it always fail. For example, the pattern

```
(a|(bc))\2
```

# kSpam

– *An open source Lotus Domino spam filter.*

always fails if it starts to match "a" rather than "bc". Because there may be many capturing parentheses in a pattern, all digits following the backslash are taken as part of a potential back reference number. If the pattern continues with a digit character, some delimiter must be used to terminate the back reference. If the PCRE\_EXTENDED option is set, this can be whitespace. Otherwise an empty comment can be used.

A back reference that occurs inside the parentheses to which it refers fails when the subpattern is first used, so, for example, (a\1) never matches. However, such references can be useful inside repeated subpatterns. For example, the pattern

```
(a|b\1)+
```

matches any number of "a"s and also "aba", "ababbaa" etc. At each iteration of the subpattern, the back reference matches the character string corresponding to the previous iteration. In order for this to work, the pattern must be such that the first iteration does not need to match the back reference. This can be done using alternation, as in the example above, or by a quantifier with a minimum of zero.

## ASSERTIONS

An assertion is a test on the characters following or preceding the current matching point that does not actually consume any characters. The simple assertions coded as \b, \B, \A, \G, \Z, \z, ^ and \$ are described above. More complicated assertions are coded as subpatterns. There are two kinds: those that look ahead of the current position in the subject string, and those that look behind it.

An assertion subpattern is matched in the normal way, except that it does not cause the current matching position to be changed. Lookahead assertions start with (?= for positive assertions and (?! for negative assertions. For example,

```
\w+(?=;)
```

matches a word followed by a semicolon, but does not include the semicolon in the match, and

```
foo(?!bar)
```

matches any occurrence of "foo" that is not followed by "bar". Note that the apparently similar pattern

```
(?!foo)bar
```

# kSpam

– *An open source Lotus Domino spam filter.*

does not find an occurrence of "bar" that is preceded by something other than "foo"; it finds any occurrence of "bar" whatsoever, because the assertion (?!foo) is always true when the next three characters are "bar". A lookbehind assertion is needed to achieve this effect.

If you want to force a matching failure at some point in a pattern, the most convenient way to do it is with (!) because an empty string always matches, so an assertion that requires there not to be an empty string must always fail.

Lookbehind assertions start with (?<= for positive assertions and (?<! for negative assertions. For example,

```
(?!foo)bar
```

does find an occurrence of "bar" that is not preceded by "foo". The contents of a lookbehind assertion are restricted such that all the strings it matches must have a fixed length. However, if there are several alternatives, they do not all have to have the same fixed length. Thus

```
(?<=bullock|donkey)
```

is permitted, but

```
(?!dogs?|cats?)
```

causes an error at compile time. Branches that match different length strings are permitted only at the top level of a lookbehind assertion. This is an extension compared with Perl (at least for 5.8), which requires all branches to match the same length of string. An assertion such as

```
(?<=ab(c|de))
```

is not permitted, because its single top-level branch can match two different lengths, but it is acceptable if rewritten to use two top-level branches:

```
(?<=abc|abde)
```

The implementation of lookbehind assertions is, for each alternative, to temporarily move the current position back by the fixed width and then try to match. If there are insufficient characters before the current position, the match is deemed to fail.

PCRE does not allow the \C escape (which matches a single byte in UTF-8 mode) to appear in lookbehind assertions, because it makes it impossible to calculate the length of the lookbehind.

# kSpam

– *An open source Lotus Domino spam filter.*

Atomic groups can be used in conjunction with lookbehind assertions to specify efficient matching at the end of the subject string. Consider a simple pattern such as

```
abcd$
```

when applied to a long string that does not match. Because matching proceeds from left to right, PCRE will look for each "a" in the subject and then see if what follows matches the rest of the pattern. If the pattern is specified as

```
^.*abcd$
```

the initial `.*` matches the entire string at first, but when this fails (because there is no following "a"), it backtracks to match all but the last character, then all but the last two characters, and so on. Once again the search for "a" covers the entire string, from right to left, so we are no better off. However, if the pattern is written as

```
^(?>.*)(?<=abcd)
```

or, equivalently,

```
^.*+(?<=abcd)
```

there can be no backtracking for the `.*` item; it can match only the entire string. The subsequent lookbehind assertion does a single test on the last four characters. If it fails, the match fails immediately. For long strings, this approach makes a significant difference to the processing time.

Several assertions (of any sort) may occur in succession. For example,

```
(?<=\d{3})(?<!999)foo
```

matches "foo" preceded by three digits that are not "999". Notice that each of the assertions is applied independently at the same point in the subject string. First there is a check that the previous three characters are all digits, and then there is a check that the same three characters are not "999". This pattern does not match "foo" preceded by six characters, the first of which are digits and the last three of which are not "999". For example, it doesn't match "123abcfoo". A pattern to do that is

```
(?<=\d{3}...)(?<!999)foo
```

This time the first assertion looks at the preceding six characters, checking that the first three are digits, and then the second assertion checks that the preceding three characters are not "999".

Assertions can be nested in any combination. For example,

# kSpam

– *An open source Lotus Domino spam filter.*

```
(?<=(?!foo)bar)baz
```

matches an occurrence of "baz" that is preceded by "bar" which in turn is not preceded by "foo", while

```
(?<=\d{3}(?!999)... )foo
```

is another pattern which matches "foo" preceded by three digits and any three characters that are not "999".

Assertion subpatterns are not capturing subpatterns, and may not be repeated, because it makes no sense to assert the same thing several times. If any kind of assertion contains capturing subpatterns within it, these are counted for the purposes of numbering the capturing subpatterns in the whole pattern. However, substring capturing is carried out only for positive assertions, because it does not make sense for negative assertions.

## CONDITIONAL SUBPATTERNS

It is possible to cause the matching process to obey a subpattern conditionally or to choose between two alternative subpatterns, depending on the result of an assertion, or whether a previous capturing subpattern matched or not. The two possible forms of conditional subpattern are

```
(?(condition)yes-pattern)  
(?(condition)yes-pattern|no-pattern)
```

If the condition is satisfied, the yes-pattern is used; otherwise the no-pattern (if present) is used. If there are more than two alternatives in the subpattern, a compile-time error occurs.

There are three kinds of condition. If the text between the parentheses consists of a sequence of digits, the condition is satisfied if the capturing subpattern of that number has previously matched. The number must be greater than zero. Consider the following pattern, which contains non-significant white space to make it more readable (assume the PCRE\_EXTENDED option) and to divide it into three parts for ease of discussion:

```
( \ ( )?  [^()]+  (?(1) \ ) )
```

The first part matches an optional opening parenthesis, and if that character is present, sets it as the first captured substring. The second part matches one or more characters that are not parentheses. The third part is a conditional subpattern that tests whether the first set of parentheses matched or not. If they did, that is, if subject started with an opening parenthesis, the condition

# kSpam

– *An open source Lotus Domino spam filter.*

is true, and so the yes-pattern is executed and a closing parenthesis is required. Otherwise, since no-pattern is not present, the subpattern matches nothing. In other words, this pattern matches a sequence of non-parentheses, optionally enclosed in parentheses.

If the condition is the string (R), it is satisfied if a recursive call to the pattern or subpattern has been made. At "top level", the condition is false. This is a PCRE extension. Recursive patterns are described in the next section.

If the condition is not a sequence of digits or (R), it must be an assertion. This may be a positive or negative lookahead or lookbehind assertion. Consider this pattern, again containing non-significant white space, and with the two alternatives on the second line:

```
(?(?=[^a-z]*[a-z])
 \d{2}-[a-z]{3}-\d{2} | \d{2}-\d{2}-\d{2} )
```

The condition is a positive lookahead assertion that matches an optional sequence of non-letters followed by a letter. In other words, it tests for the presence of at least one letter in the subject. If a letter is found, the subject is matched against the first alternative; otherwise it is matched against the second. This pattern matches strings in one of the two forms dd-aaa-dd or dd-dd-dd, where aaa are letters and dd are digits.

## COMMENTS

The sequence (?# marks the start of a comment which continues up to the next closing parenthesis. Nested parentheses are not permitted. The characters that make up a comment play no part in the pattern matching at all.

If the PCRE\_EXTENDED option is set, an unescaped # character outside a character class introduces a comment that continues up to the next newline character in the pattern.

## RECURSIVE PATTERNS

Consider the problem of matching a string in parentheses, allowing for unlimited nested parentheses. Without the use of recursion, the best that can be done is to use a pattern that matches up to some fixed depth of nesting. It is not possible to handle an arbitrary nesting depth. Perl has provided an experimental facility that allows regular expressions to recurse (amongst other things). It does this by interpolating Perl code in the expression at run time, and the code can refer to the expression itself. A Perl pattern to solve the parentheses problem can be created like this:



# kSpam

– *An open source Lotus Domino spam filter.*

At the end of a match, the values set for any capturing subpatterns are those from the outermost level of the recursion at which the subpattern value is set. If you want to obtain intermediate values, a callout function can be used (see below and the `pcrcallout` documentation). If the pattern above is matched against

```
(ab(cd)ef)
```

the value for the capturing parentheses is "ef", which is the last value taken on at the top level. If additional parentheses are added, giving

```
\( ( ( (?>[^( )]+) | (?R) )* ) \)
  ^          ^
  ^          ^
```

the string they capture is "ab(cd)ef", the contents of the top level parentheses. If there are more than 15 capturing parentheses in a pattern, PCRE has to obtain extra memory to store data during a recursion, which it does by using `pcre_malloc`, freeing it via `pcre_free` afterwards. If no memory can be obtained, the match fails with the `PCRE_ERROR_NOMEMORY` error.

Do not confuse the `(?R)` item with the condition `(R)`, which tests for recursion. Consider this pattern, which matches text in angle brackets, allowing for arbitrary nesting. Only digits are allowed in nested brackets (that is, when recursing), whereas any characters are permitted at the outer level.

```
< (? ( (?R) \d++ | [^<>]*+ ) | (?R)) * >
```

In this pattern, `(?R)` is the start of a conditional subpattern, with two different alternatives for the recursive and non-recursive cases. The `(?R)` item is the actual recursive call.

## SUBPATTERNS AS SUBROUTINES

If the syntax for a recursive subpattern reference (either by number or by name) is used outside the parentheses to which it refers, it operates like a subroutine in a programming language. An earlier example pointed out that the pattern

```
(sens|respons)e and \1ibility
```

matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If instead the pattern

```
(sens|respons)e and (?1)ibility
```

# kSpam

– *An open source Lotus Domino spam filter.*

is used, it does match "sense and responsibility" as well as the other two strings. Such references must, however, follow the subpattern to which they refer.

## CALLOUTS

Perl has a feature whereby using the sequence (`{...}`) causes arbitrary Perl code to be obeyed in the middle of matching a regular expression. This makes it possible, amongst other things, to extract different substrings that match the same pair of parentheses when there is a repetition.

PCRE provides a similar feature, but of course it cannot obey arbitrary Perl code. The feature is called "callout". The caller of PCRE provides an external function by putting its entry point in the global variable `pcre_callout`. By default, this variable contains `NULL`, which disables all calling out.

Within a regular expression, `(?C)` indicates the points at which the external function is to be called. If you want to identify different callout points, you can put a number less than 256 after the letter C. The default value is zero. For example, this pattern has two callout points:

```
(?C1)\dabc(?C2)def
```

During matching, when PCRE reaches a callout point (and `pcre_callout` is set), the external function is called. It is provided with the number of the callout, and, optionally, one item of data originally supplied by the caller of `pcre_exec()`. The callout function may cause matching to backtrack, or to fail altogether. A complete description of the interface to the callout function is given in the `pcrecallout` documentation.

Last updated: 03 February 2003

Copyright © 1997-2003 University of Cambridge.