How to Use Regular Expressions

An Introduction into Regular Expressions

A Tutorial for TheBat!-Users

by

Gerd Ewald (Author)

and

Marck D. Pearlstone (Editor)

© Gerd Ewald, Ratingen 2002-2003

Table of Contents

1. Introduction....4
2. Regular Expressions....4
   2.1. What does "Regular Expression" mean?....4
3. Simple Patterns....5
   3.1. Simple Known Characters....5
   3.2. Search Patterns for Metacharacters....6
   3.3. Simple Unknown Characters....6
   3.4. Groups of Characters and Character-Classes....6
   3.5. Overview and Summary....7
4. Complex Patterns....8
   4.1. Line Boundaries....8
   4.2. Word Boundaries....9
   4.3. Alternatives....10
   4.4. Special Character Groups and Classes....10
   4.5. Overview and Summary....11
5. Special Elements - Part 1....12
   5.1. Quantifier....12
   5.2. Grouping of Elements, Subpattern and Quantifiers again....14
      Grouping of Elements....14
      Subpattern....15
      And Quantifier again....16
   5.3. Overview and Summary....19
6. Special Elements - Part 2....22
   6.1. Assertion....22
   6.2. Backreference....23
   6.3. Conditional Regular Expressions....24
   6.4. Options, Modifier....25
   6.5. Specials....27
      Meta-Characters....27
      Square brackets....27
      Numbers and Digits....27
      Restrictions when using Regex....28
   6.6. Overview and Summary....28
7. How to use Regular Expressions in TB....31
   7.1. Macros....31
   7.2. Other Possibilities to Use Regular Expressions in TB....35
   7.3. Overview and Summary....37
8. Final Conclusion....39
9. List of Elements of Regular Expressions and Macros....42

10. Examples for various regular expressions....45
Check mail-address....45
Check for valid IP-address....45
Check for multiple matching recipients in an email address....45
Check for multiple matching domain names in email address....46
Conditional check for digits in the subject line....46
Check for several consonants in a row....46
11.Credits....47

1. Introduction

Whenever I came across something interesting in a mail that was created with TheBat! like "cleaned" Subject-strings or automagically deleted PGP-lines, I would ask in one of the mailing lists: "How did you do that?". Quite often I would receive a reply like "You will need a regex for that!" And sometimes the result was something like:

```
%QUOTES="%SETPATTREGEXP=""(?is)(-----BEGIN PGP SIGNED.*?\n(Hash:.*?\n)?\s*)?
(.*?)(^(- --|--\n|-----BEGIN PGP SIGNATURE)|\z)""%REGEXPBLINDMATCH=""%text""%
SUBPATT=""3"""
```

This is only a simple example of those cryptic looking combinations of TB!-Macros and regular expressions which are simply called "regex" by the TB-experts. To me it seemed a random sequence of characters; as if a cat walked across my keyboard. Awkward, arbitrary and cryptic, that at least was my impression until Januk Aggarwal (special thanks to him) gave me a short introduction to regex at TBTECH and my workmate Alfred Rübartsch gave me a copy of Jeffrey Friedls excellent book "Mastering Regular Expressions".

Although I entered the fascinating world of Regular Expression with the help of these two, I am still not an expert in the "regexian" language. Anyway, as an advanced beginner, I have dared to write this tutorial to hopefully explain some things and give a good start in "Regular Expressions" to other beginners.

This tutorial is meant to bring you into closer contact with the regex topic. Well, let's see how it works; let's see whether we will be able to explain the "regex"-example above by the time we come to the end of this tutorial.

2. Regular Expressions

2.1. What does "Regular Expression" mean?

Regex are not only used in TB! You can find them in quite a lot of different UNIX-tools (e.g. grep), in some programming languages like PERL (Practical Extraction and Report Language, sometimes called 'Pathologically Eclectic Rubbish Lister' <bg>) and even my editor UltraEdit uses them.

Laura Lemay wrote in her book "PERL in 21 days" that the term "Regular Expression" makes no sense at first sight (to be honest: even at second sight it still makes no sense to me), because these are not real expressions and furthermore no one really can explain why they are "regular"! Well, let's ignore this; let's simply accept that the term "Regular Expression" has its origin in formal algebra and that they are indeed part of Mathematics.

The easiest and most convenient way to define "Regular Expression" is to say: "They are search patterns to match characters in strings."

Those of you who have tried to find files using the DOS command line or the search function in the Explorer may have used patterns like:

```
dir *.doc
```

```
copy *.??t c:\temp
```

These examples show patterns that consist of letters, stars, question marks and other characters to define which files should be listed or copied. In the first example only files that have the suffix "doc" should be listed. In the second example only files that have a three-letter suffix and a "t" as last character in the suffix should be copied.

But these regex are merely wildcards! In no way as mighty as "Regular Expressions". One can't compare them to real regex, which offer much more than wildcards for characters.

3. Simple Patterns

To explain some regular expressions and to understand the examples given in this tutorial we have to define how the regex will appear. I will envelope the regular expression in quotation marks ("). If you want to test the regex you will have to copy the part between the "-characters. Testing regular expressions? Yes, sure, this is possible.

You have to download a DLL written by Dirk Heiser

(http://www.Dirk-Heiser.de/RegExTest/RegExTest_V0.3beta.zip)

and copy it into your TB-directory. Then, when you open the TB-help, you will find a tabfolder called RegEx. Or, if you are using the CHM-Version of the help (this probably applies only to the German version), you can use this DLL by creating a link on your desktop which opens the DLL:

```
"%windir%\system32\rundll32.exe <your_path>regextest.dll, Run"
```

Please, I really recommend that you download this utility. It will make it so much easier to follow the tutorial.

3.1. Simple Known Characters

Ok, let's start with simple search patterns: "`give or take`"

Yes, you won't believe it, this is already a regular expression: it matches the string 'give or take' in a text. Exactly these characters! And no, this does not mean that this pattern matches either 'give' or 'take'. The regular expression only matches if the characters in quotation marks appear somewhere in the text!

Regular expressions are stubborn and stupid: they will look for exactly what they are told to search for. They are case sensitive and they are not interested in word boundaries unless told to be so. For example, our first regex will find the characters in the following string: 'You have to for**give or take** the consequences!'

3.2. Search Patterns for Metacharacters

Regular expressions can search for any character – alphanumeric, hexadecimal, binary numbers, etc..

A small but important exception are those characters that have a special meaning in regular expressions, the metacharacters.

Metacharacters are:

`* + ? . ( ) [ ] { } \ / |^ $`

(Hi experts: Yes, you are right! I stretched the truth!! These are not actually all metacharacters. But trust me, just assume that I am right for now. We will see later why I prefer to define the above as metacharacters).

I will explain these metacharacters later in the tutorial, step by step, as many of them as necessary. Just one thing for now. If you want to search for those characters as they stand you have to tell the regex that you want to do so. The regular expression has to be told that you don't mean to use a metacharacter but want to search for it literally. So you have to "escape" or "mask" the character with another character (which of course is a metacharacter in itself <g>): it is the backslash "\"

If you want to match a question mark the regex has to be "`\?`". If it is a slash you're after, you have to enter "`\/`". And, although it looks queer, if you want to find a backslash you need to type two of them "`\\`"

3.3. Simple Unknown Characters

The first metacharacter we are going to learn is a dot "`.`" It represents exactly one unknown character we want to match, no matter what this character might be (Hello experts: let's come to exceptions later. Ok?)

"`M.ller`" will match 'Miller', 'Meller' or 'Millerton' but not 'Milton Keynes'. "`h..s`" matches 'hips' or 'hers'. And within the word 'house', the same regex will match 'hous'.

Later we will learn about some more metacharacters; ones that will allow us to look for more than one unknown character without repeating the dot over and over.

3.4. Groups of Characters and Character-Classes

Some metacharacters define groups of characters, making a very powerful tool. There is a wide variety of these groups. Let's start with the easy ones:

"`\d`" symbolizes a digit. "`\d\d`" searches for any sequence of two digits.

"`\w`" stands for any letter or the underscore character. This group is called 'alphanumeric characters'.

With what we already know we can create our first more complicated looking regex:

"`Re \[\d\]:`" searches for the string 'Re' followed by a space, an opening square bracket, any digit, a closing square bracket and finally a colon in a text. Ooops, that looks like a Subject-line which was created by someone who forgot the %SINGLERE in his reply template ;-)

There are -of course- metacharacters which have the opposite meaning: "`\W`" and "`\D`"

\W is stands for any non-alphanumeric character and \D means any character that is not a digit.

Another elegant method to define your own group of characters is to use the square brackets [ ] which stands for 'character classes'. With square brackets, the regex will search for exactly one character, no matter how many characters are in between these brackets: "`[AEX]`". This combination will match any one-letter string that must be one of A, E or X.

You may even define ranges of characters. You don't have to type in every character of the range, no; regexian makes it easy for you: just enter the first character of the range, a hyphen "-" and the last letter: "`[e-z]`" means that all letters from e to z should be matched. "`[AEXe-z]`" is a combination of both: a one-letter string with one of A,E,X or any letter within the range e to z.

This is a powerful tool in regexian: "`[0-1][0-9]\/[0-3][0-9]\/`" will match only a MM/DD/ formatted date. Other combinations which are not a date (e.g. 35/47/) won't be found. (Yeah, you're right! My regex will match 19/39/ which isn't a terrestrial date at all. We will get this one later once we have learned some more elements....)

You can negate character classes with one keystroke. Just add a "^" after the opening square bracket and that's it. 'Find any character as long as it isn't 1,2,3 or 4!' in regexian is: "`[^1-4]`". Oh, we should remember this one for later. This funny '^' character has a totally different meaning when not in square brackets!

3.5. Overview and Summary

What did we learn in this chapter?

- regular expressions search for any character. "er" looks for the exactly these letters in that order. All regex are case sensitive unless told not to be so.
- Regexes use characters with a special meaning: metacharacters. To find them literally they must be escaped. This is done with a preceding backslash: `* + ? . ( ) [ ] { } \ / | ^ $`
- a dot "." is used to a single unknown character. It is a metacharacter.
- There are metacharacters which symbolize groups of characters like

  \d for digits ([0-9])

  \D for non-digits ([^0-9])

  \w for alphanumeric characters ([a-zA-Z0-9_])

\W for non-alphanumeric characters ([^a-zA-Z0-9_])

- It is possible to define your own set of character-classes by using square brackets e.g. "[A-Z]". A ^ as first character in the square bracket negates the class.

Exercises:

What does each regex match?

```
"\d\d\.\d\d\.\d\d\d\d"
"\w\w\w, \d\d \w\w\w \d\d\d\d"
".. \[[0-9]\]:"
"[a-zA-Z]"
```

First example: first it will match two digits. Next comes the backslash and a dot. That means, the dot is escaped and is no metacharacter. So the two digits has to be followed by a dot. Again two digits and a dot. And finally four digits! This is the European format of a date DD.MM.YYYY

In the **second example** the regex searches for three alphanumeric characters followed by a comma, a space, two digits, another space. Next come three alphanumeric characters, a space and finally four digits. Phew, what could this be?

Well, it looks like a format for dates again, but this time in an Anglo-American format: Tue, 19 Feb 2002. Well, like the first example, this regex is not perfect. It only matches dates with two-digit days. We will see later how we can modify the regex to find one- or two-digit days

Third pattern: the regex looks for two characters and a space. The next character is a square bracket. Then a square bracket follows which isn't escaped by a preceding backslash: this defines group of characters! Any digit in the range 0 to 9 is going to match the string. A square bracket again and a colon. This combination would match 'Re [2]:'

In the **last example** the regex looks for only one character. Any letter is allowed, even capital letters. Why isn't "\w" used? Well, that would include the underscore and perhaps the author doesn't want to match that character ;-)

4. Complex Patterns

Ok, that was an easy start! But it wasn't very interesting, was it? But if simple search patterns were all that "Regular Expressions" offer, it wouldn't be worth a tutorial.

So, there has to be more! Okay, let's get going with the more complicated stuff:

4.1. Line Boundaries

Instead of having a regex look for text anywhere in the string we can force it to search in specific parts of the string. These "anchored" patterns have their own metacharacters: ^ and $

The circumflex ^ means that the search pattern is anchored to the start of the line; the dollar $ means that the regex will look for the pattern at the end of a line (Yes, dear experts, for now, let's take a string as one line. Ok?)

Example: "`^give or take`" This pattern will only be matched if 'give' is at the beginning of a line and is followed by 'or take'.

Or: "`This is the end$`" is only matched if it appears at the end of the line. It doesn't matter what comes first: 'This is the end' has to be the end of the line!

You can use these two metacharacters to speed up the regex. I admit, it is not all that important when you use regex in TB! because you won't be working with large amounts of data. But on the other hand: it can't hurt anyone ;-) Why does the regex work faster if you use the circumflex or the dollar, you ask? Ok, let's use our example regex "`^give or take`" on the string 'Once upon a time': the regex machine checks whether the first thing it finds is the beginning of the line. This returns TRUE. Next it checks the following character whether it is a 'g'. The search process is cancelled at once because this returns FALSE! Now what would have happened without the circumflex? The regex machine would have checked the second, third, fourth etc. character to match the search pattern, only to find out that the search pattern doesn't exist in that string. The longer the string, the more time the regex machine takes to fail ;-)

4.2. Word Boundaries

But there is more that regexian offers. Word boundaries! Some people forget about this because they think there is another way to define word boundaries. Believe me, there is, but it's nowhere near as easy as this!

"`\b`" makes the regex searching for the pattern at word boundaries: "`\bgive or take`".

Hey, we know this one, don't we? That is our first example again! The pattern that was found in 'You have to forgive or take the consequences!' but now won't be found thanks to the word boundary metacharacter.

I remember a discussion in one of the German TB-lists where someone asked why this metacharacter is necessary, because a word could be recognized by surrounding spaces. This is not a good idea: words could end at question marks, exclamation marks, a full stop.... A regex like "`ain`" would indeed match 'Again a good idea' but wouldn't find 'Oh no, not again.' You can avoid that when you use "`\b`" instead.

Of course, this metacharacter can be negated, as can the others: "`\B`" which means that the regex should match characters everywhere in a string other than at word boundaries.

Another example should explain this: "`Re\B.`" The regex has to match the characters 'Re' as long as they are not a word boundary, followed by any other character (the dot). Now, we have the string: 'Re: or Reply:'. Try it in the regex tester. What happens? The result is 'Rep'. Replace \B by \b and the regex matches 'Re:'. Everything clear now?

4.3. Alternatives

You remember the first example in this tutorial "`give or take`"? When I introduced it I made the redundant remark that this regex wouldn't match 'give' OR 'take'. Well, this remark wasn't really redundant: I needed something to start this chapter, some kind of transition <bg>. Because this is the chapter that explains how we can use the OR; how alternative patterns are defined.

To search for alternative patterns, regexian offers a special metacharacter: it is the vertical bar or may be better known as pipe-symbol "|". So, what would have been necessary to search for 'give' or 'take'? "`give|take`". The regex checks whether it matches 'give'. If not it checks the string for 'take'.

What happens if the string contains both alternatives? Well, to be honest, when I started with regex I was convinced that the first alternative in the regular expression would be matched. But no! The regex will match the alternative that comes first in the string! Let's get into details with an example:

Given the regex "`this|the|that`" and the string 'the hand that signed this paper' (Ok, ok. You didn't really expect sample strings from Shakespeare or Yeats, did you?) What does the regex return? 'the' is the answer! Try it in the regex-tester!

You may combine alternatives as you have seen in the last example. Just have a look at the following "`^re:|^aw:|^fwd:`". This means that in all three alternatives the regex has to match the beginning of the line first. Some characters follow and each alternative ends with a colon. Yep, you are right: there must be a way to simplify this one. And like in Mathematics you can use brackets to make the regex shorter "`^(re|aw|fwd):`".

Well, those simplifications do not necessarily make it easier to read:

"`th(is|e|at)`" would be a correct and simple alternative to the first example in this chapter but it is not exactly an easy-to-read example. ;-)

4.4. Special Character Groups and Classes

We have already introduced some of the special search patterns for groups and classes of characters. I would like to present some others with varying significance.

In almost every real regex you find the character class "`\s`" . It represents so-called whitespace characters, that is any character which produces white space on the screen: space, tabs, newline, carriage return, line feed. It's ok if you just remember that any void space in a string will be matched. And, of course, you may negate this pattern: "`\S`" matches any character that does not appear as white space in the string.

"\A" is a seldom used search pattern: it matches the beginning of the string. This is not the beginning of the line; no, to search for that we would have used ^. Later when we talk about options like multiline you will see where you can use this one. "\Z" is related to "\A": \Z matches the end of the string and again I can only say: "This is not end of line" because that would have been $. You will see the difference when we talk about options. Sorry, but you have to be patient :-)

4.5. Overview and Summary

This chapter explained some more possibilities in defining search patterns:

- line boundaries are matched by circumflex ^ (beginning of a line) and dollar $ (for end of a line).
- Word boundaries are matched by "\b". It searches for characters that appear at the beginning or the end of a word. "\B" represents characters that do not appear at the end or the beginning of words.
- Search patterns can contain alternative characters to match. The alternatives are separated by a vertical bars "|". Characters that appear at the same place in each alternative can be placed before or after brackets that enclose the alternatives. "^(Re|Aw|Fwd)" All alternatives must appear at the beginning of the line in a string to be matched.
- Spaces or tabs are so-called whitespace characters for which a special search pattern exists: \s The negation is \S
- Beginning and end of a string: \A and \Z

Exercises:

1. Given the regular expression "(R.:$|^R.:)" and the string 'Ra: or Re:'. What does the regex match?

2. I want to match 'Re:' at the beginning of a line even if it comes with a reply counter e.g. 'Re[2]:'. With what we have learned about regular expressions so far: what is the regex for doing that?

3. Let's try to DIY a regex that matches 'Re' at the beginning of a line or ')' at the end of a line.

4. What do these search patterns mean?

   a) "^"

   b) "^x$"

   c) "^$"

First exercise: the regex matches 'Ra:'. We expected that, didn't we? The regex matches the alternative which comes first in the string.

Ooops, the solution of the **second exercise** already looks quite professional, doesn't it: "^(Re|Re\[\d\]):" Ok, may be you have something different; something that looks a bit simplified like: "^Re(|\[\d\]):". It is a good example because simplified version shows an absolute void as the first alternative in the brackets - the '|' symbol has nothing to the left of it other than the open bracket that starts the "sub-string".

Third exercise:

"`(Re|\)$)`" is one solution. You didn't forget to escape the bracket, did you? Fine, well done *g*. Now, if you can, try this one in the regex-tester with the following string: 'Re[2]: bladibla (was: more bla)'. You will see that the regex exactly matches just 'Re' because at this point the regex machine returned TRUE for the match. If the beginning of the string is changed to something else only then will the regex match the bracket.

Fourth exercise:

The first pattern searches for any text that has the beginning of a line or that starts at the beginning of a line. This would include any text - even a void line would be matched.

The second pattern just looks for a single x character that is alone in a line.

Last, but not least, the third pattern: it searches for lines that have a beginning and an end, but nothing else: these are void lines!

5. Special Elements - Part 1

Everything we've had so far hasn't been too difficult. But this chapter is heavy stuff. Please, do me a favour: read this chapter carefully. Be patient! Try everything with the regex tester; get familiar with the elements in this chapter: they are the essential for creating proper regex. Although this may be a bit more complicated than the chapters before, it is certainly more interesting ;-)

5.1. Quantifier

We already know to define patterns for matching single characters, groups of characters, character classes or ranges of characters. We can use alternatives in our search patterns. But something of absolutely vital interest is missing - the ability to define repetitions.

You remember the example that was a regex to search for the European formatted date:

"`\d\d\.\d\d\.\d\d\d\d`"

For every single digit we wrote "\d". Isn't there another way, much simpler than repeating the metacharacter as often as the regex wants to find the character? Yes, there is! There are quantifiers!

`+ * ?` are the most important quantifiers.

The "+"-character means that the character preceding the plus-sign has to appear at least once at the specific point of the string. "`fo+l`" matches 'fool', 'fol' and 'foooool'.

"Re:\s+", for example, means that at least one whitespace has to follow 'Re:' to be matched.

I hear some of you experts: yes, the usage of quantifiers is not only restricted to characters. You can use them to repeat metacharacters, character classes and some other elements we are yet to learn. ;-)

The star "*" represents any number of occurrences of the preceding character at the specific point in the string. 'Any' really means 'any', even if the character doesn't appear at all. Ooops, what's the use of that?

Well, let's have a look at the following example: "Re:\s*\w+"

Huh, that already looks as cryptic as those regex the experts use <g>. What does this regex mean?

Search for a 'Re' followed by a colon. Then any number of whitespace characters may appear - even no spaces at all. What for? In proper subject lines there should be a space. But imagine we would like to match any subject string even if someone modified it manually and deleted the space. We have to tell the regex that there might or might not be a space. Anyway, both possibilities should be found. This can be done with the star as quantifier. Well, finally, there has to be at least one alphanumeric character.

Caution: the meaning of this quantifier is sometimes misinterpreted. Look at the following task: a regex has to be defined that matches only lines of a string with only digits in it. One solution I saw was: "^[0-9]*$"

But this regex matches void lines as well; the star stands for 'no digit' as well as for 'any digit'. So the regex machine returns TRUE when no digit is in a line. If you want to make sure that there is at least one digit in a line you have to use the plus-sign: "^[0-9]+$".

The question mark means that the preceding character may appear once or not at all at the specific point of the string. A bit like the star only that the number of occurrence has the maximum '1'. "h..?s" matches 'hers', 'hips' and 'his' or 'has'. Within 'house' it matches 'hous'; within 'hose' it matches 'hos'.

There is another way to define repetitions: "{x,y}" This is a way to explicitly define how many repetitions of the preceding characters you want. In this formula 'x' denotes the minimum number and 'y' the maximum number necessary for the preceding character. "\d{2,4}" means that only two to four digits in a row are matched.

If you omit the second number 'y' but leave the comma in the curly brackets "{x,}", then there is no upper limit and the minimum is x-times the preceding character. "\w{3,}" matches any string with at least three word-characters.

If you omit not only the second number but the comma as well "{x}", then this means the exact number of appearances of the preceding character. "\d{6}" matches exactly six digits. This quantifier gives us a new way to write our regex that matches European formatted dates : "\d{2}\.\d{2}\.\d{4}"

The three quantifiers I introduced at the beginning of this chapter are simply special ways to write one of the following regex:

{0,1} = ?

{1,} = +

{0,} = *

Before I can tell you more about quantifiers and what has to be kept in mind when using them, I have to introduce parentheses (round brackets) as a grouping device.

5.2. Grouping of Elements, Subpattern and Quantifiers again

Grouping of Elements

In the chapter about alternatives, the parentheses crossed our way for the first time. They were used as they are in maths: common parts of the pattern are written outside the round brackets.

Now we will learn something new: we can use the parentheses to group parts of the regex to be dealt with as a single element of the pattern. A following quantifier is applied to the grouped part of the regex. E.g.: "`foo(bar)?`" matches 'foo' and 'foobar'

Another example:

"`Re\s*(\[\d+\])?:`" There it is again, the reply counter in a subject line. This time it looks already quite professional. First of all we look for 'Re'. After any number of whitespaces (or none at all) digits in square brackets may follow. This part is grouped. Finally there has to be a colon.

Let's have a closer look at the regex: why is it defined in that way?

First the whitespaces: we don't know whether the author of the subject line inadvertently added one or more spaces after the 'Re'. Even if he did nothing and left the string untouched we want the Regex to match the string. Well, I agree, there shouldn't be any space, but you never know … ;-) That's why we use "`\s*`" at this point.

Then the digits in square brackets: we allow any number of digits in the square brackets by using the plus-sign as quantifier. But there has to be at least one digit! Because there is no upper limit for this character, the way to infinity is free <vbg>.

Finally the counter '[#]' itself: this part is grouped. This element need not appear in the string to result in a successful match. That is why we use the question mark.

The regex therefore will match:

'Re:'

'Re [1]:'

'Re[123]:'

It will not match 'Re[]:'. Something to think about and to try on your own: what has to be changed so that the regex matches this one?

Ok, here is the solution: replace the '+'-sign in the square bracket with a star: "`Re\s*(\[\d*\])?:`"

Within 'Re [1]: [3]:' it matches 'Re [1]:'. It does not match the second reply counter. Ok, if we want to find such awful subject lines we have to work on our regex a bit more: it should match any number of counters that may have colons and -you never know - that may or may not be followed by spaces. Finally the last character has to be at least one colon: "`Re\s*(\[\d+\]:*\s*)*:+`"

Well, it is possible for a subject to begin like that although there is only a small probability that it will really happen. I can envisage many of combinations of reply counters. The regex does not match all of them. If you want to have the regex match other combinations, go ahead, try it! Test it with a regex of your own making, but: there is one major point you should keep in mind. There is no perfect Regex. The more you try to improve the regex to match even more possibilities and combinations of characters, the more complicated the result will be. You will have to pay for this kind of perfectionism: either you won't be able to read your regex anymore or the Regex will become buggy whenever you make even the smallest change to it. It is easier to live with some erroneous matches and to sort them out manually than to create the perfect Regex. Jeffrey Friedl published a regex to match email-addresses in "Mastering Regular Expressions": it is more than 6000 bytes. It was a good example of being too perfect, as he stated.

Ok, back to the job-in-hand: let's have another example of how to group elements. We had a pattern to match European formatted dates: "`\d{2}\.\d{2}\.\d{4}`" As you can see, the beginning "`\d{2}\.`" is repeated. Right, so this can be simplified: "`(\d{2}\.){2}\d{4}`" The first part, now grouped in parentheses, has to appear twice. This is for example '01.02.'. This is not an optimal version of the search pattern: day and month numbers still have to be two digit numbers and silly values for both are still allowed. But wait; you will get your chance. Let us learn some more elements before you are given the job of optimising the pattern in an exercise <g>.

Subpattern

Grouping with parentheses has another effect in regexian that is widely used in a lot of regular expressions in TB. Characters that were found due to a grouped pattern or element are stored in a temporary variable for further use. These variables are known as a subpattern (SubPatt in TB). We should have a look at an example to help us understand that:

'bill.doors@macrohard.com'

We use the regex "`(\w+)\.(\w+)@.*`". The first parentheses matches 'bill', the second one 'doors'. These two are each now stored respectively in subpattern 1 and subpattern 2.

Or:

"(\d+\.)(\d+\.)" When the string is '22.05.' then '22.' is stored in subpattern 1 and '02.' in subpattern 2.

How do I find out which is the first subpattern? Well, in our simple examples it is obvious: everything that is matched by the first pair of round brackets goes to subpattern 1, the second pair returns subpattern 2, etc

But what if the regex looks like: "Re\s*(\[(\d+)\])*:" The part that is enclosed by the first opening bracket and its corresponding closing bracket is stored in subpattern 1. The part that is enclosed by the second pair starting at the second opening bracket is stored in subpattern 2. With 'Re [4]:' our example would result in:

Subpattern 1 = '[4]'

Subpattern 2 = '4'

Important: each opening bracket creates a new variable or subpattern. If you want to avoid this, you have to insert "?:" just behind the opening bracket: "(?:…)" is a grouping which does not store the matched string in a subpattern.

What does the regex-machine store in a subpattern when a quantifier is applied on a grouped element? Example: "(\d{2}\.){2}\d{4}"

If the string is '23.05.2002' the first pattern is matched at '23.'. Now the regex machine goes on to find the same pattern in the string a second time. If successful the matched characters are stored in the same subpattern. In other words: the second match overwrites the first one. In our example the subpattern will show '05.'

The regex-tester shows the contents of each subpattern: with every subpattern it will offer another tab panel. That one with '0' on it shows the whole match, while that with '1' on it shows the match of the first subpattern, etc.

And Quantifier again

Ok, now let's move on to some special behaviour relating to quantifiers, Some of them have a 'human' peculiarity: they are greedy! You don't believe that? Well, look at the following string <g>:

"The abbreviation 'ISP' stands for 'Internet Service Provider'."

We want a regex that finds the text that is enclosed by inverted commas and stores it in a subpattern:

"(.*)'(.*)'.*"

Nothing difficult really: find everything that comes before an inverted comma, then everything in between and finally everything that follows…

And? Did you try it on the regex-tester? What is in subpattern 2? "Internet Service Provider". Ooops, I expected "ISP" because it comes first in the string. :-o It is quite obvious that the first group (.*) greedily matched most of the string and left only what was

at least necessary for subpattern 2 to match the whole string. Furthermore, the last element ".*" in the regex allowed 'nothing' or void to follow. Keeping this in mind: this part leads to a successful match even if nothing is to be matched. The star stands for as many appearances as there are or none at all!

Ok, here's another example:

We want to extract as many parts of an email-address as possible. We've already got a solution for the first part, the name; but that wasn't a good one because it only allowed word characters. We have to make this more generic. Let's take (.*) for the first part. The second part is some text delimited by a dot. But this may appear more than once before the @-sign ends the name section. The Regex should therefore find the following examples of addresses:

'1234abc@mail.com'

'1234.abc@mail.com'

'12-34.abc.def@mail.com'

So, the regex starts with "`(.*)\.?(.*)*@`". After that any text may follow, possibly delimited by more dots. We will ignore this for the example and go for extracting only that text that comes last after the last dot, so that the regex does not get too complicated. This should be done with "`(.*)\.(.*)`"

"`(.*)\.(.*)*@(.*)\.(.*)`"

What do we expect in the subpatterns when '12-34.abc.def@mail.com'

Subpattern 1 = '12-34' ?

Subpattern 2 = '.abc' or '.def' or 'abc.def' ?

Subpattern 3 = 'mail' ?

Subpattern 4 = 'com' ?

Ask the regex-tester:

Subpattern 1 = '12-34.abc'

Subpattern 2 = ' def '

Subpattern 3 = 'mail'

Subpattern 4 = 'com'

Subpattern 1 contains almost the all of the first part, subpattern 2 only the last three characters before the @. Of course, we expected that, didn't we? We already know that the star is greedy: it stored as many characters as it could into the first subpattern.

Caution: not only stars, I mean star-signs are mean and greedy <vbg>, the plus-sign is as well! Don't forget that!

Let's take another string to test the regex: '12-34.abc.def@mail.test.com'. Now the star in the third parentheses "`(.*)`" is greedy and 'eats' almost everything after the @ up to the last dot, storing 'mail.test' and not 'mail'.

How can we avoid that? We are going to learn another meaning of the question mark (Calm down, this is only the second one. There are many more to come and you will eventually come to understand why a regex is full of these funny question marks ***g***): just add a question mark to the greedy pattern and you make the pattern less greedy.

Let's do that. We add a ?-sign to the first pattern:

`"(.*?)\.(.*)@(.*)\.(.*)"`

Subpattern 1= '12-34'

Subpattern 2= 'abc.def'

Subpattern 3= 'mail.test'

Subpattern 4='com'

For a better understanding I shall try to explain what the regex-machine does: the regex-machine does not restrict the greediness of the (.*). In the moment it discovers the pattern (.*?) the following happens: it stores as much as possible into this subpattern. Then it steps back one character at a time to find a point where a successful match is found.

I'm going to explain it using our example regex "`(.*?)\.(.*)*`" and the string '12-34.abc.def'. The Regex machine stores '12-34.abc' into the first subpattern. This is the maximum that the Regex allows because a dot and some text follow this string. But now the machine realizes that there is a question mark, which suppresses the greediness of the first subpattern. Thus, it steps back one character before the 'c' and checks whether or not the Regex leads to a successful match. No, it does not. So, again, take one step back and a check again. Still no hit. Back again to a position before the 'a'. And now the machine realizes that this would lead to a successful hit because of the preceding dot. The machine takes the position exactly before the first dot. In reality, it would have to do some more back-stepping to find out that this position is the last one possible with the minimum of characters for a successful match. But I reckon we've looked deep enough in to the way it works for now.

Back to our first example where we wanted to match text between inverted commas. The regex was "`(.*)'(.*)'.*`" and the text "The abbreviation 'ISP' stands for 'Internet Service Provider'." Let's alter the Regex to "`(.*?)'(.*?)'.*`"

Both grouped elements need a question mark otherwise "ISP' stands for 'Internet Service Provider" would be stored in the second pattern. To add a question mark in the second element alone wouldn't help very much because the first (.*) remains greedy.

5.3. Overview and Summary

This was a quite difficult section. Not only for you to read and understand. No, it was even difficult to write and create the text, from which I hope you got some idea. This section covers one of the basic elements of regexian that you will need in every Regex.

The following elements were presented:

- Characters that repeat preceding characters are called quantifiers:

  + the preceding character must appear at least once

  ? the preceding character may appear once or never

  * the preceding character may appear in any amount of times or never

- There are quantifiers that allow to define exact ranges of the frequency of the preceding character:

  {x,y} the preceding character has to appear at least x-times but not more than y-times. One may omit parts of the range: {x,} stands for at least x-times with no maximum. {x} means exactly x-times.

- Parentheses are used to group multiple character sequences into patterns so that we can apply quantifiers to them. "(ab)+" means that the combination of 'ab' has to appear at least once to be matched.
- Patterns in parentheses are stored in variables for further use. These variables are called subpatterns in TB. In the case of multiple parentheses where groups are grouped, the outer subpattern contains all inner subpatterns. Furthermore, the first opening round bracket creates the first subpattern, the second defines the second subpattern and so on.
- Quantifiers with no upper limit may be greedy in some search patterns. + and * after a dot make the regex take in as much as it can to lead to a successful match. (.*)(.*) will include the whole match in subpattern 1 and nothing in subpattern 2.
- A greedy pattern can be made ungreedy by adding a question mark to it (.*?) In the first step it still will match all that is possible but then it will do some backstepping to give back one character at a time until the minimum characters that constitute a successful match are reached.

Exercises:

1: The last regex we created for searching European format dates was: "`(\d{2}\.){2}\d{4}`" It wasn't perfect because it didn't allow single digit days or months nor two digit years to be matched (D.M.YY or any other combination). That's worth an making into exercise, isn't it?

2: You've got the solution for question 1? Ok, that solution is quite interesting but now we can try to write an improved Regex for matching European formatted dates.

If possible we would like to allow only combinations of digits that look like a terrestrial date. Well, we do not want to exaggerate: it's ok if the Regex matches February, 29th (29.02.) even if it isn't a leap year ;-).The only important points are: it should be in the format DD.MM.YYYY or D.M.YY or any combination and it should be restricted to dates that exist.

3: Imagine you receive bug-reports via an on-line system. The reports are standardized and all have the same format (more or less). We need a regex that extracts the more important information. The reports look like:

Sender: firstname.lastname@agency.com

Date: TT.MM.JJJJ

Report-no.: xyz123

Please try to define a regex that extracts the following parts into subpatterns: first name, last name, agency, date, report-no.

4. Write a regex that matches the time in the form hh:mm:ss. Make sure that only valid combinations are returned.

Problem 1:

```
"\d{1,2}\.\d{1,2}\.(\d{4}|\d{2})"
```

You created something else? Doesn't matter, it may be a correct solution: there is often more than one way to do it!

```
"(\d?\d\.){2}(\d{4}|\d{2})"
```

is in my opinion an elegant solution. A not so good idea is something like `"\d{2,4}"` for matching the year: it allows three digit years.

Problem 2:

This is a bit tricky. In these cases I like to divide the problem into smaller chunks. Which days are possible:

a) 01-09, the preceding zero could be missing.
b) 10-29, all months of a year have at least 29 days. Ok, there is one error we are allowed to make: February only has 29 days in leap years. We will assume this is ok, otherwise it might be almost impossible to create the Regex.
c) 30, all months except February
d) 31, only January, March, May, July, August, October, December.

Possible numbers for months are 01-10 (the preceding zero might be missing) and 11, 12. We want to allow two or four digit years. In case of four digit years we only accept those that start with 19xx or 20xx

Ok, now we have what we need. Let's start:

Case a) and b) combined with the allowed months gives us:

"(0?[1-9]|[12][0-9])\.(0?[1-9]|1[0-2])\."

Case c) with all possible months:

"30\.((0?[13-9])|(1[0-2]))\."

And finally case d) with possible months:

"31\.(0?[13578]|1[02])\."

Now the years:

"(\d{2}|(19|20)\d{2})"

The first three parts have to be alternatives whereas the pattern for years is mandatory. To avoid that the Regex matches within a longer sequence of digits to find something that only looks like a date, we envelope the whole Regex with \b metacharacters. That should give

```
"\b(((0?[1-9]|[12][0-9])\.(0?[1-9]|1[0-2])\.)|(30\.((0?[13-9])|(1[0-2]))\.)|
(31\.(0?[13578]|1[02])\.))(\d{2}|(19|20)\d{2})\b"
```

[Note: the regex is wrapped due to layout reasons. All must be used as a single long line!]

Incredible: that's a cracker! You found something different? Even something better? Well, I think that is 'normal'. You can always write a regex in another way to give the same result. And of course: you can improve almost every Regex. My Regex only shows one way to approach the problem: the way I like to do it. I hope you were able to follow my thinking.

Problem 3.

This is not very difficult. Again, divided into chunks of the whole problem:

First name and last name can be extracted from the mail-address. "Sender:\s*(.*?)\.(.*?)@(.*?)\.\w+\s*" should be sufficient.

The question mark in the second subpattern might be redundant because the @-character follows anyway. But it won't hurt anyone, would it?

Date: phew, we are in luck. The format is mandatory. We don't have to use the killer regex of problem 2 ;-):

"Date:\s*((\d{1,2}\.){2}\d{4})\s*"

And now the report number:

"Report-no.:\s*(.*)"

To make sure that the regex checks the whole string we add \A at the beginning and \Z at the end.

```
"\ASender:\s*(.*?)\.(.*?)@(.*?)\.\w+\s*Date:\s*((\d{1,2}\.){2}\d{4})\s*Report-
no.:\s*(.*)\Z"
```

[Note: the regex is wrapped due to layout reasons. All must be used as a single long line!]

Subpattern 1,2,3,4 and 6 will contain the information we wanted.

Problem 4.

I think we have already had some practise at dividing bigger problems into smaller ones. The time-problem is another one. It should be mere routine now. And, it is much easier than it looks at first sight, because the format is fixed!

Hours are from 00 to 19 and 20 to 23 (24 equals 00!!):

`"([01][0-9]|2[0-3]):"`

Minutes and seconds have the same format and the same combinations of digits, 00 to 59:

`"([0-5][0-9]:){2}"`

Altogether, enclosed by word boundary (\b) metacharacters:

```
"\b([01][0-9]|2[0-3]):[0-5][0-9]:[0-5][0-9]\b"
```

6. Special Elements - Part 2

Did you think chapter 6 would be more complicated than chapter 5? No, calm down: chapter 6 will deal with some elements that may be a bit more complex but on the whole these elements are not too difficult to learn.

6.1. Assertion

When I wrote the German version of this chapter I had something to start with: what does assertion mean? This question was possible because there is no German word for this aspect of the terminology of regular expressions: it's simply called 'assertion'. Even a look into Friedls book didn't help: he doesn't use this expression. He calls this element of regexian "lookahead". Ok then, let's look at what an assertion can do for us in regular expressions.

An assertion can be used to find out whether characters precede or follow the matched part of a string. Well, that's nothing new. We could do that without an assertion. But: the assertion checks the characters without including them in the match: it does not "eat" the characters.

Let us explain this with an example:

We want to match the string 'foo' within a string only if it is followed by 'bar'. But do not want 'bar' to be part of the match! Without an assertion we would have used "foobar", but this regex matches 'foobar'. If we use an assertion "(?=" the regex looks like "`foo(?=bar)`" The match now is 'foo'.

Of course there are negative look-aheads or assertions "(?!" too.

Ok, let's try this one on our more or less senseless example: "`foo(?!bar)`" means that 'foo' will be matched except when 'bar' follows but only 'foo' is the resulting matched string. So it matches 'foo' within 'foolish' while 'foobar' doesn't match at all.

But be careful, there is a trap: one might think that "`(?!foo)bar`" only matches 'bar' if 'foo' does not precede it. Wrong!! This regex matches 'bar' in any event with no exception. This assertion is a look-ahead one that only looks ahead to find 'foo' ahead of the point where 'bar' is found. But there will always be a 'bar' and never a 'foo'. Ok? Did you understand this?

What we need is a look-behind assertion! And, as if by magic, here is one: "?<=" is a positive look-behind assertion and "(?<!" is the negative version.

Example:

"`(?<!foo)bar`" matches 'bar' if 'foo' does not precede it

"`(?<=foo)bar`" matches 'bar' only if 'foo' precedes it.

You can use assertions in alternatives, e.g.: "`(?<=proba|possi)bility`"

So, 'bility' is matched if either 'proba' or 'possi' precedes this. But it would match 'bility' as well if 'impossi' preceded the string. But I think you expected that ;-))

There is a restriction to the search patterns allowed in assertions: the length must be absolute, which means, you can't use quantifiers. "`(?<=\d+,)\d\d`" would produce a syntax error.

The different branches of alternatives in assertions may have different lengths, but they have to be predefined!

Furthermore: different (but still defined) lengths of branches are permitted at the top level: "what's that?" Have a look at the example:

"`(?<=ab(c|de))`" is not permitted, "`(?<=abc|abde)`" is permitted. The second opening parenthesis makes us leave the top level, causing the assertion to become unpredictable for the regex machine.

You may use assertions in a sequence and you may nest them:

Example: "`(?<=\d{2}\.\d{2})(?!00\.00)\s+Payment`" matches 'payment' if preceded by any amount as long as the amount is not '00.00'

Or: "`(?<=(?<!im)possi)bility`" matches 'bility' only if preceded by 'possi'. But it won't match if preceded by 'impossi''.

6.2. Backreference

In an earlier chapter we learned something about subpatterns. These were characters that were stored in some kind of variables. I wrote "... are stored in a temporary variable for further use…". Well, now we will see what 'further use' means:

Let's take a regex to explain what I mean: "`(sens|respons)e and \1ibility)`"

This regex will find either 'sense' or 'response'. Whatever it matches is stored in the first subpattern (you remember - the first opening parentheses?). Then it has to be followed by ' and '. Next comes "\1". This means: use the content of the first subpattern as part of the search pattern. Because this is followed by 'ibility' the regex matches either 'sensibility' or 'responsibility' whatever was found at the beginning of the string. So 'sense and sensibility' or 'response and responsibility' would have a successful match but never 'sense and responsibility'.

Some restrictions:

the backreference must not appear in the subpattern it is related to: "`(a\1)`" would never give a positive match. On the other hand, if the subpattern is followed by a quantifier, it is allowed: "`(da|de\1)+`" This matches 'dadadada' or 'dadeda' or 'dadedadadada'.

6.3. Conditional Regular Expressions

This is an element that is not very common: conditional regular expressions. The principle is: "If pattern A is found, look for pattern B; if not then look for pattern C".

The correct syntax is "(?(condition)Yes-Pattern|No-Pattern)" or "(?(condition)Yes-Pattern)"

But there is a restriction to the condition pattern: it has to be either a sequence of digits or an assertion.

What is it good for? Let's assume we have to extract a date from some text. But for whatever reason it could be a European DD.MM.YYYY or English DD, MMM YYYY formatted date. We only know that at the beginning of the line there is either 'Datum' for the European (German) version or 'Date' for the English one and the date terminates the line.

What we want is a regex that matches the English formatted date if it is preceded by 'Date', otherwise it should match the European formatted one:

```
"(?(?=^Date)Date:\s(\d+),\s([A-Za-z]{3})\s(\d{4})$|Datum:\s(\d{2}\.)(\d{2}\.)
(\d{4})$)"
```

[Note: the regex is wrapped due to layout reasons. All must be used as a single long line!]

At the beginning of this chapter I told you that the condition has to be either an assertion or a sequence of digits. But these digits must be backreferences. The condition wouldn't literally search for digits. So what does it all mean?

Example: we receive mails with 'Name' in the first line followed by a value that is that name. The name may change between mails, but we know, that the name will occur again within the mail, and when it does it is related to an attribute we want to extract, let's call it the shoe size

"`Name:\s*(.*)?$`" will find the name. This is followed by something we are not interested in. But then the name appears again followed by a colon and the shoe size, which are digits:

"`.*?(?(1):\s*(\d+))`" Both parts combined:

"`Name:\s*(.*)?$.*?(?(1):\s*(\d+))`"

Have a try with the following text:

'Name: James Herriot

Bladibla

James Herriot: 9'

(Note: if you use the regex tester you have to switch on the Singleline option. We will learn about that in a while)

6.4. Options, Modifier

We've had quite a lot of new vocabulary to learn in regexian. Now let's learn something about modifiers. What do they do? Well, they modify something. But what? Elements of regexian are modified by modifiers. I can hear you shout:"Oh no; after I've learned all that about regex and so many different elements ". Don't worry: I am not going to explain all the possible modifiers or options; we will restrict ourselves to those that are essential and most important.

Look at the following options:

i for Caseless

The regex machine is forced to ignore the case of letters. It will search for the pattern ignoring the case (upper/lower) of the pattern

m for Multi-line

The regex machine usually takes the string as a whole line, no matter whether there any newline characters (\n). The circumflex “^” that indicates the beginning of a line only matches the beginning of the string and the dollar “$” matches the end of the string instead of matching the end of a line or a terminating newline at the end. Go ahead, test it with the regex tester: uncheck the Multi-line option. Then enter the following wrapped text:

'This is a test that

has several

lines with a test

at the end of a line’

and the regex "`test$`". Nothing will be matched. Switch multi-line on and 'test' will be matched.

When this option is active the regex machine will indeed recognize each newline character; the text now consists of multiple lines. This is important when we are going to check the entire text of a message in one hit.

s for DotAll

As we learned in one of the first chapters, the dot matches any character other than the newline character. Once this option is set, the dot matches newlines as well.

But this is not actually the whole truth: the newline will also be matched by all negated character classes that do not include the newline, e.g.: "[^x]" matches everything except the character 'x': that includes any newline.

x for Extended

When this option is enabled the regex machine ignores any whitespace character in a search pattern. Thus you are now able to include remarks in the regex, wrapped in #-characters.

To search for whitespaces in this mode you have to escape them "\ " or you use "\s".

Furthermore you can define your own character class that searches whitespaces e.g.: "[ ]"

If you have a look in the regex tester's options menu you will find some more options or modifiers. I don't want to explain them all. There are some special options that are explained in books or other tutorials. They are not really necessary for a basic understanding of regular expressions. The four I explained above will be useful to you and sufficient for most purposes.

How do we switch them on? That is easy: you just enter the letter that indicates the option in parentheses in which a question mark precedes the letter: "(?" and ")". E.g.: "(?i)" switches on 'ignore case'. You may combine several options for example: "(?im)" means 'Caseless, Multi-line'. Furthermore you may switch options on or off:

"(?im-sx)" switches caseless and multi-line on and Dotall and extended off.

If a characters appear before and after a "-"-character then the option is switched off. You may use the options anywhere in the regex. They may appear at the beginning as well as in the middle.

"(?i)Test" is the same as "Te(?i)st". In the case where an option is switched on more than once in the top level part of the regex than the machine will use the option that comes last in the search pattern. Although you may enter the options at any point in the regex I recommend that you do it at the beginning.

Well, there is no rule without an exception: if an option appears within a subpattern it will only apply to the subpattern: "(a(?i)b)c" matches 'abc' as well as 'aBc'.

Something to think about:

"(a(?i)b|c)" is the regex. Does it match a 'C' or only a 'c'? It is obvious that it matches 'aB'….

6.5. Specials

Ok, let's finish this chapter with some special elements and rules that will cross our path in TB only every now and then. I don't want to go into details; this chapter is more like a glossary to look up if a regex "behaves" oddly.

Meta-Characters

In the first chapter of this tutorial I mentioned the meta characters and listed] and }. These two aren't actually meta characters. You may recall that I asked you to assume they were. If you searched for them as literals you wouldn't need to escape them. But, I always escape them to keep my regex easy to understand. I avoid errors caused by misunderstandings, which I will elaborate here:

Within a character class defined by "[" at the beginning only the following characters are meta characters:

\ to escape

^ to negate a character class but only if the circumflex is the first character to appear within the class

- to indicate a range

] to terminate the character class definition

Ok, now let's have a closer look to the following more or less senseless regex:

"`[Y-]345]`" I wanted to define a range within the class that includes 'Y' to ']' and the digits 3,4 and 5. But what happens? Does the regex match 'Z34' or parts of it? No!

Instead try 'Y345]' or '-345]'. And here we are, it is matched. The only problem is... that is not what we wanted.

Ok, I am going to explain what happened: the first close square bracket is interpreted as the end of the character class. The regex matches strings beginning with 'y' or '-' followed by '345]'. Yes, "`[Y-\]345]`" is the correct solution. What have we learnt? Although "]" is not a metacharacter outside a character class it's a good idea to escape them every time one searches for them as literals.

Square brackets

Let's have a closer look at square brackets and special cases. Assuming that caseless is switched on then "[aeiou]" will match 'A' as well as 'a'. But "`[^aeiou]`" will match 'A' only when caseless is switched off.

Numbers and Digits

We use \d to try to find decimal digits. Of course it is possible to look for characters using hexadecimal or octal character codes. The regex "`\x09`" matches the character with the hexadecimal code 09.

Octal numbers are a bit more difficult : the syntax is quite easy "`\ddd`", where each d is a digit. The regex searches for the character with an octal code of 'ddd'.

Or, and now it gets a bit tricky, for a backreference. The regex machine takes any number lower than 10 as a backreference if the number is not inside a character class. Inside a character class or if there are not enough parentheses to define a relative subpattern the number is taken as a pattern for octal codes.

Examples:

\040 is octal 'space'

\40 is octal unless there are enough (more than 40) parentheses defining subpatterns

\6 is always a backreference

\11 could either be a backreference or a 'tab'

\011 is always a 'tab'

\113 is always octal, because there are no more than 99 backreferences allowed

And what is \0113??

Restrictions when using Regex

This is just to inform you about restrictions that we have to bear in mind when using regular expressions. You and I, as 'normal' users, won't reach these limits of regexian but a Regex must not exceed 65535 bytes. There are no more than 99 subpatterns allowed. The total number of elements - like groups, assertions, options and conditionals - must not exceed 200. Furthermore the length of the entire text that is checked for the pattern is restricted as well, but we won't reach this limit in TB. It is restricted to the value of the system's largest positive integer. Because the Regex machine needs to reserve storage for subpatterns and for quantifiers with undefined length due to recursive processing the maximum length available will be reduced. But, to be honest, this is not really of interest to TB-Users ;-)

6.6. Overview and Summary

This chapter explained some of the special features of regular expressions. We should know enough about regex by now to be able to use them in TB's macros.

Short summary:

- an assertion can check whether characters appear in before or after a search pattern without including these characters in the match. These are:

  positive lookahead-assertion (?=

  negative lookahead-assertion (?!

  positive lookbehind-assertion (?<=

  negative lookbehind-assertion (?<!

You are not allowed to use quantifiers on them - that would make them unpredictable to the regex machine.

- Strings that are matched as subpatterns are available for further use within the same regex. These backreferences can be addressed by "\#" where # is a positional number indicating the parenthesis pair that defines the subpattern
- Assertions or Backreferences may be used to create conditional regex "(?(condition pattern)yes-pattern|no-pattern)"
- The vocabulary of regexian can be modified using modifiers or options. They may precede the Regex in parentheses "(?modifier)". We discussed a few of them here:

  i for Caseless

  m for Multi-line

  s for DotAll

  x for Extended
- We learned that some characters become metacharacters in character classes and behave differently.
- A regex can search for hexadecimal or octal character codes. Remember though that there is a possible conflict with backreference numbers.
- The length of a regex is limited because the result has a limited length. Even the text to which a Regex is applied must not exceed a certain length. This should only be of minor interest when using regex in TB.

And here are the **exercises**:

1. Try to define a regex that recognizes doubled words (e.g.: 'the the') Don't forget that the second word may appear at the beginning of the next line. The Regex should only match words and not parts of words (not 'the theme') and it should ignore case.

2. Ok, now let's try to write a simple version of a subject cleaning regex. 'Re' can be followed by anything and a colon. Then the original subject appears. After that a space could follow and a former subject enveloped in parentheses introduced with 'was:' follows. We would like to extract the original subject. This is a very simple version of a cleaner.

3. We receive mail with order amounts of some product. We need the integer of these amounts (without the decimals). The amounts may be mailed in EUR or $. Well, the problem is that the symbol for the decimals is different in both systems. Furthermore, the sign indicating Thousands is different as well: #,###.##$ or #.###,##EUR. The regex should know which of the versions it has to match.

1:

The most important hints were the word wrapping (multiple lines) and caseless. These are options which we switch on: "`(?im)`".

Finding words should be easy: "[a-z]+" We only look for words without digits. We do not have to define capital letters because we already switched "caseless" on. But we only want to search for whole words: so we have to use \b in front of the pattern. We can't use \b at the end of the word, because it is okay to end a sentence with a word and start the next sentence with the same word. So we need to allow at least one whitespace to follow. "(?im)\b[a-z]+\s+" This is the beginning: we still need something to find the second appearance. We have to store the result of the first match in a variable to have a backreference. Ok, second try:

"(?im)(\b[a-z]+)\s+\1"

But, this matches 'the theme', which we didn't want to have matched. But this is easy now: after the second appearance nothing but a word boundary may follow and that's it:

```
"(?im)(\b[a-z]+)\s+\1\b"
```

2: This is a very simple subject cleaner.

What we defined above as a possible subject should look like 'Re[2]: proper subject we want ;- (was: old subject)'

The beginning of the subject can be matched by "^Re(.*?):" - 'Re' at the beginning of the line, followed by anything or nothing if there is no counter. We could have done this with ".*". But unfortunately this is greedy and might match more than we want.

The original subject can be found after the colon; to make sure that we don't extract redundant whitespace we match them first. "^Re(.*?):\s*(.*?)" The original subject is stored in subpattern 2. Again there is a question mark to avoid greediness. What is missing? Ah, something that matches the old subject: otherwise it would be included in subpattern 2.

A whitespace and then an opening round bracket follows. Then, as we said, it is introduced with 'was'. But this old subject could be missing from time to time so we can't insist on its appearance: "\s*(\(was:.*\))*$"

Or as a whole:

```
"^Re(.*?):\s*(.*?)\s*(\(was:.*\))*$"
```

We use the "$"-character to make sure the regex reads the whole line. We can do that because we may expect that the subject is a single line. Those of you who aren't sure should use "\Z" for end of string instead.

3: Ok, this is an exercise that looks quite artificial. But sometimes one needs stupid examples to clarify something (I remember some exercises in physics when I studied that assumed "one-dimensional cattle" or "weightless Christmas bulbs". These weren't much cleverer than my example ;-))

I agree that this could be done using a different regex but I wanted a conditional one:

First of all we need an assertion that looks for something, two digits and a dollar sign: "(?=.*\d{2}\$)" If this exists, it should be the $-version: "([\d,]+)\.\d{2}\$" I simplified

the problem and defined a character class that allows only digits and commas. (Well, here a mismatch is possible when there is an arbitrary string with digits, several commas then a dot and two digits followed by a dollar sign. Hmmm, well, ok, you're so clever? You improve it! ***g*)**

If there is no $-version the Regex should match the EUR-version: "`([\d\.]+),\d{2}EUR)`", which uses the same simplification as above.

The full regex should look like:

```
"(?(?=.*\d{2}\$)([\d,]+)\.\d{2}\$|([\d\.]+),\d{2}EUR)".
```

Did you notice that the EUR-result is stored in the second subpattern while the $-version is stored in the first one?

7. How to use Regular Expressions in TB

Finally, we can try to use our new language in TB. First of all we have to know which tools are available to work with regular expressions. These tools are TB's macros.

7.1. Macros

Not all of TB's macros support the use of regex. Most of the macros have nothing to do with regex, but you can use regex on them to extract or modify the information.

And that is one feature of TB that makes it so powerful.

The first macro we will look at is: %REGEXPTEXT="regex" What does it do? It searches for the pattern "regex" within the original text of a mail and returns the matched characters. The syntax is quite straightforward, look at the following example:

```
%REGEXPTEXT="[\d\.]+"
```

This macro used in a quick template and applied to a mail returns digits and dots.

Let's have a look at a fairly similar macro: %REGEXPQUOTES="regex"

This macro does exactly the same as the first one except that the returned text is not plain text but quoted text.

That was nice and easy. But when it comes to the extraction of text from the header of a mail (kludges) or address book entries we need to combine some macros:

The first one we will need for that is %SETPATTREGEXP. It is used to define the search pattern in the way %SETPATTREGEXP="regex". "regex" is the regular expression you created to match the text.

The second one is %REGEXPMATCH. Again, this is easily defined: %REGEXPMATCH="string" with "string" being any text. It can be a template, which means that any generic text can be used, so almost any TB macro can be used to provide the text here.

The definition of a regex through %SETPATTREGEXP is valid unless it is overwritten by a second appearance of a %SETPATTREGEXP. This means you can use the same pattern on several different generic texts in one go.

Before we have a look at another example I have to correct something. Did I say the syntax is quite easy earlier in this chapter? Well, that's true as long as one only looks at one macro. But let's see how this changes when we let the macro parse some text:

We already know the macro %REGEXPQUOTES. This could be written in a different way. Let's assume that we receive Mails from a feedback form. Part of the content is "newsletter: yes" or "newsletter: no". We would like to create an autoresponder that uses exactly this information in a reply template, for example:

"Thank you for filling out our feedback form. You entered 'newsletter: yes/no'. Are you sure?"

You can create more sophisticated text and a better filter to use different templates for the reply, but for the moment let's stick to this example;-).

The macro %QUOTES defines what text is to be used as quoted text in a reply. The only problem is that we have to tell %QUOTES which text should be used. After that we can copy it to the reply template, add our standard text and save it.

Ok, first the regex: "`^newsletter:\s*(yes|no)`". This has to be defined by

```
%SETPATTREGEXP="^newsletter:\s*(yes|no)".
```

We already know that %REGEXPMATCH applies the search pattern on any generic text, so we need a macro that provides the original text of the mail and that is %TEXT. Now we have to put it all together and create a template that uses the macros in the correct order.

The only thing that makes it difficult to use these macros are the "-characters which are used as delimiters for the definition part. In %SETPATTREGEXP the search pattern is defined between these and in %QUOTES the text that will be inserted as quoted is defined. Once you start to combine the macros you have to tell TB which "-character is delimiter of which macro: the first macro must know whether the second "-character is the end of the macro or the beginning of the second macro. The same applies at the end of the second macro and so on. This can be achieved by doubling the "-character (escaping) or using different delimiters.

Simply, this looks like:

%M1="%M2=""Def2""%M3=""Def3""". This is getting a bit confusing and hard to follow, so we could instead say:

%M1="%M2='Def2'%M3='Def3'". The example above would look like:

```
%QUOTES="%SETPATTREGEXP='^newsletter:\s*(yes|no)'%REGEXPMATCH='%TEXT'"
```

This example could be written in a simpler way:

```
%REGEXPQUOTES="^newsletter:\s*(yes|no)"
```

but this is because we extracted text out of the original text with %TEXT.

Next comes a macro combination that allows the extraction of several parts of the text. We know that we could define subpatterns in the regex by grouping sections with parentheses. We must now find a way to address them within TB.

TB provides a macro for this %REGEXPBLINDMATCH="string". But this does not return anything useful. Of course, we wanted to extract parts of the text not the whole text itself. So we still need a macro that allows us to tell the macro which of the subpatterns are to be used. And this is %SUBPATT="n". 'n' denotes the n-th subpattern in the regex.

Now this combination will be quite difficult to read and understand. So I will explain it using an example and will generate the whole macro combination bit by bit. After that I will combine everything.

From the original date of a mail we want to extract the year, two digits only, and use it as quoted text. The date is provided by %ODATE. The regex is "`\d{2}(\d{2})\b`". That means we want to extract only two digits if they are preceded by two digits and followed a word boundary. Thus the first macro is:

```
%SETPATTREGEXP="\d{2}(\d{2})\b"
```

The text that is used to find the date is defined using the macro % REGEXPBLINDMATCH="%ODATE". We are looking for the first subpattern, so % SUBPATT="1".

Now we put all together, we don't forget to use the alternate '-characters:

```
%QUOTES="%SETPATTREGEXP='\d{2}(\d{2})\b'%-
%REGEXPBLINDMATCH='%ODATE'%SUBPATT='1'"
```

[Note: the regex is split using the %- macro and can be entered as two lines!]

Another example? There is a regex for reply templates that modifies the name of the recipient. Instead of 'Gerd Ewald' we would like to have 'Gerd Ewald at TBUDL…..' Well, we could download this regex somewhere, but let us try to create it ourselves.

%OFROMNAME will give us the name.

The reply address is given by %OREPLYADDR. We will extract the list's name with a regex. Usually the name of the list precedes the @-character: %SETPATTREGEXP="(.*?)\@"

This is used in combination with %REGEXPBLINDMATCH="%OREPLYADDR" of which we only want subpattern one : %SUBPATT="1"

The result is then the contents of the TO-field. Watch out, before you can enter text this field has to be cleared. This is done by an initial assignment which is void.

```
%TO=""%TO='"%OFROMNAME at %-
%SETPATTREGEXP=_(.*?)\@_%-
%REGEXPBLINDMATCH=_%OREPLYADDR_%-
```

```
%SUBPATT=_1_" <%OREPLYADDR>'
```

[Note 1: the regex is split using the %- macro and can be entered as seen!]

[Note 2: the regex makes use of a feature of recent versions of TB where any character may be used as a quoting delimiter, in this case the underscore and single quote as well as double quote. Users of earlier versions will have to resort to using the clumsier double delimiter syntax]

The original reply address has to be added enclosed in “<>”-characters at the end.

As you can see, the syntax is quite easy and stereotypical. The only difficult thing is to find out which macro provides the necessary information and how to extract it with the regex.

Here another example that is available at Marck's FAQ-page (www.silverstones.com)

```
%WRAPPED='Historians believe that on %ODATE%-
%SETPATTREGEXP="(?m-s)Date\:\s*?((.*?[\d]{4})\s*?([\d]{0,2}\:%-
[\d]{0,2}\:[\d]{0,2})\s*?(.*))"%-
%REGEXPBLINDMATCH="%HEADERS" , at %SUBPATT="3"[GMT%SUBPATT="4"]%-
(which was %OTIME where I live) you wrote:'%-
```

Here, once again, the %- macro is used to make the whole combination easier to read.

This has no special meaning except that it tells TB that the following line should be treated as a continuation of the first line. The %WRAPPED means that the result of the macro combination will be word wrapped at the defined column in TB.

What does the macro do?

The first part "%WRAPPED='Historians believe that on %ODATE%-" is just some kind of a link up: on every reply the date of the original mail should be added to the text 'Historians believe that on '.

The second part contains the regex that is much more interesting to us (I deleted the %- macro to show the regex in one line):

```
"(?m-s)Date\:\s*?((.*?[\d]{4})\s*?([\d]{0,2}\:[\d]{0,2}\:[\d]{0,2})\s*?(.*))"
```

The option multiline is switched on and DotAll is switched off: (?m-s)

Then the regex looks for ‘Date:’, which may be followed by any number of whitespaces. Due to the greediness of the star a question mark follows. The author escaped the colon with a backslash that isn’t necessary. I don’t know why he did that but it won’t cause problems, so we'll leave it alone.

Now the first parenthesis follows. There is no need to group this part and I assume it is done for easier reading. You may delete it but then bear in mind that the total number of subpatterns has changed.

The second parenthesis looks for anything that consists of four digits. We know that the regex will look in the kludges (%HEADERS) for the date. So we guess that the author will look for something like ‘year’. This may be followed by whitespaces.

Now we come to the third parenthesis. This is the one the author needs. He searches for three numbers with zero, one or two digits. These numbers are separated with colons.

That is obviously the time. Whitespace may follow and with the fourth subpattern all of the rest is matched: this is nomore than the GMT-information.

A closer look on the regex shows that it is applied to the header lines and only that only subpattern three and four are really needed.

The result could be:

'Historians believe that on Sonntag, 7. April 2002 , at 11:22:59[GMT

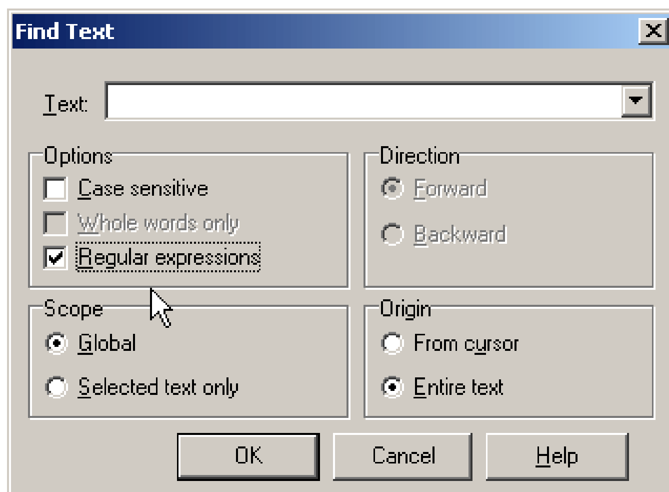
+0200](which was 11:22 where I live) you wrote:'

It works although the layout would need a bit DIY.

7.2. Other Possibilities to Use Regular Expressions in TB

There are other possibilities for using regex in TB than macros.

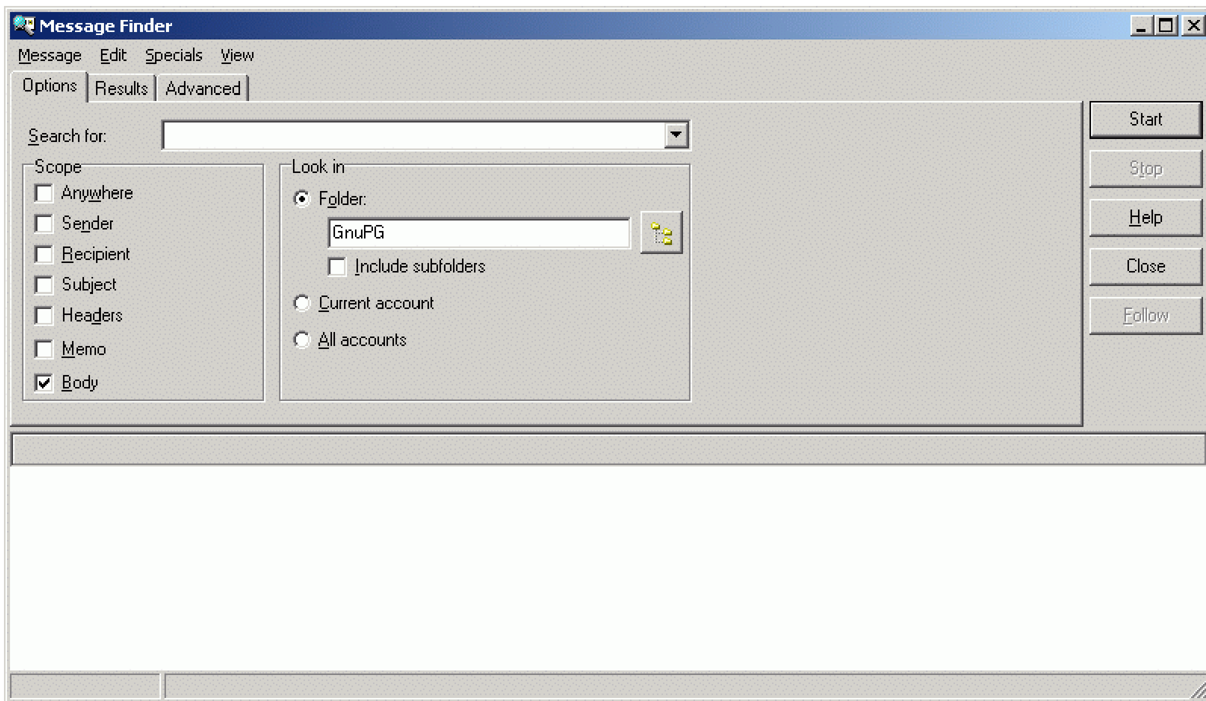
For example the text search option for in the mail editor. It is especially useful to search for strings in long mails with the special features that regex offers.



Picture 1 : Find Text

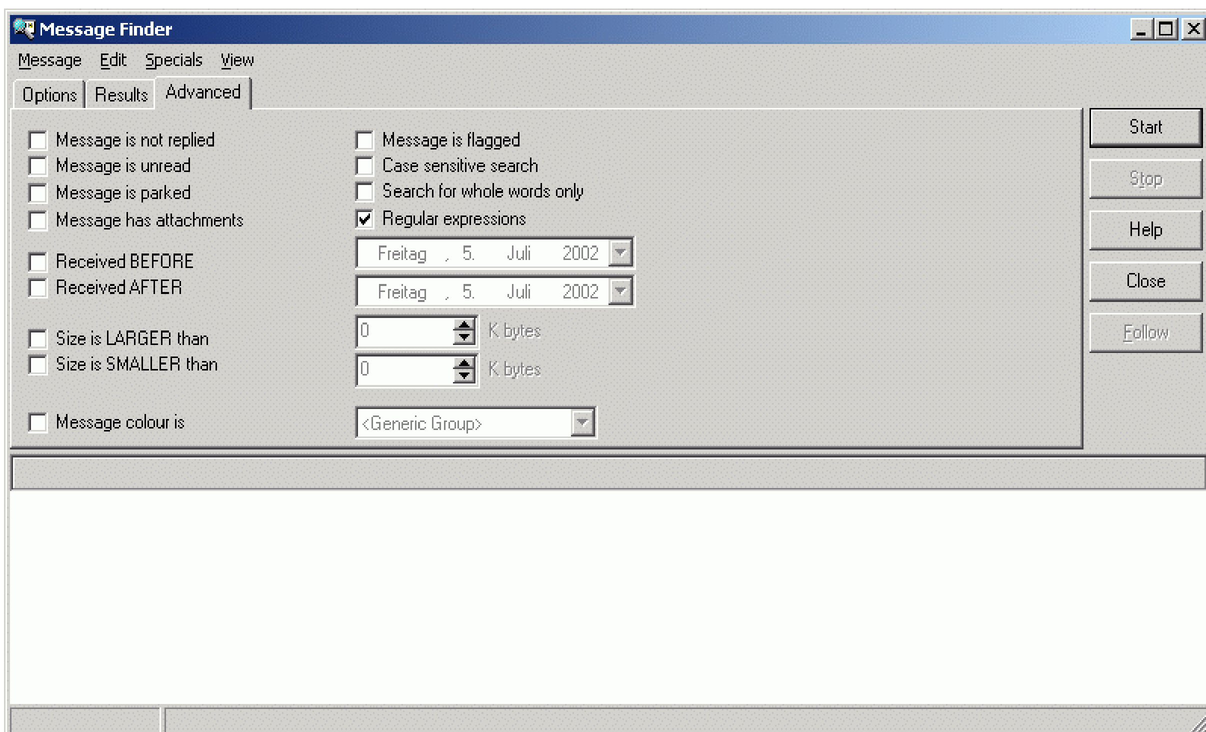
This window may be opened with Ctrl-F or using the ‘Edit Find’ menu entry in the mail editor. Just enter the regex in the text line. Don’t forget to check the ‘regular expressions’ box in the Options section.

In almost the same way I can search for text within stored mail, I can search text mails in folders using regex. Just press F7 while in folder view. This opens a search window, which offers the facility to search for text in mails. In the ‘Options’ tab panel you can enter the regex in the ‘Search for’ field.



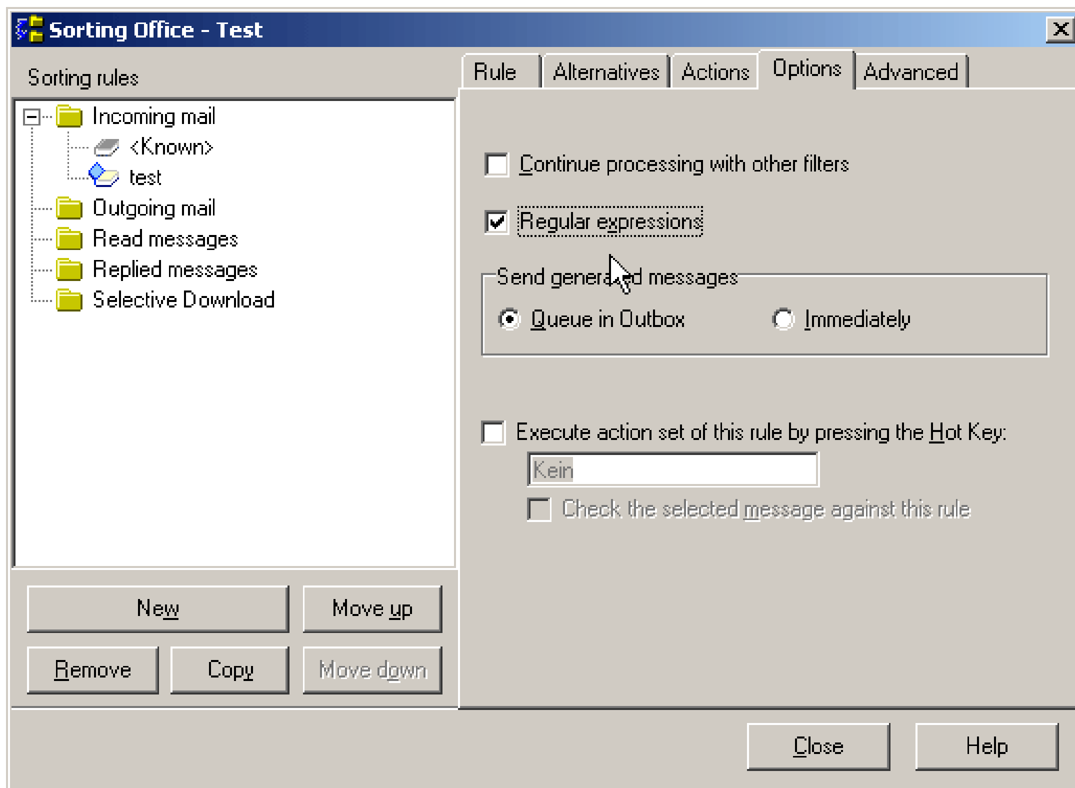
Picture 2: Message Finder - Options

Go to the ‘Advanced’ tab panel and check “Regular Expressions”.



Picture 3: Message Finder - Advanced

You can use regex in filter conditions to optimise the organisation of your inbox. This is a field where regex are as efficient as in macros. Go to the 'Account, Sorting Office/Filters' menu item. Open the filter definition, go to the 'Options' tab panel and check 'Regular Expressions'.



Picture 4: Sorting Office

7.3. Overview and Summary

What did we learn in this final chapter?

There are several ways to use regular expressions in TB, which are:

- TB offers macros that can use regular expressions to find, extract and modify mail text:
    - %REGEXPTEXT="regex": returns the matched string within a mail as text
    - %REGEXPQUOTES="regex" returns the matched string within a mail as quoted text
    - %REGEXPMATCH="string" defines the generic text in which the regex should match the specified string and return the matched text. Any macro or text may be used for 'string'

  - %REGEXPBLINDMATCH="string" is used to define the generic text in which the regex should match the specified string. It does not return any text. The %SUBPATT is needed to return the text. Any macro may be used for ‘string’.
  - %SETPATTREGEXP="regex" defines the regex for %REGEXPMTACH and %REGEXPBLINDMATCH. The definition is valid if not overridden by a subsequent %SETPATTREGEXP
  - %SUBPATT="n" returns the n-th subpattern when used with % SETPATTREGEXP and %REGEXPBLINDMATCH
- You can use regular expressions to look for specific messages as well as to search strings within mails. Furthermore you can use them for defining filters.

Exercise 1:

You remember the regex we wrote to clean the subject line?

```
"^Re(.*?):\s*(.*?)\s*(\(was:.*\))*$".
```

Try to improve this one: instead of ‘(was:xyz)’ PGP-users will find ‘(PGP Decrypted)’. The regex should find these kinds of subject as well. Furthermore the regex should be available within a reply template.

Exercise 2:

In the last chapter I described a macro that modifies the TO-address for mailing lists:

```
%TO=""%TO='"%OFROMNAME at %-
%SETPATTREGEXP=_(.*?)\@_%-
%REGEXPBLINDMATCH=_%OREPLYADDR-%-
%SUBPATT=_1_" <%OREPLYADDR>'
```

Try to change it in such a way that it is no longer necessary to use % REGEXPBLINDMATCH and %SUBPATT but %REGEXPMATCH. You will need to modify the regex. Hint wanted? Ok: The subpattern was created because otherwise the @-character would have been included in the match. The only thing you have to do is to find a regex that does not match the @-character and has no subpattern.

Solution 1:

Well, that is not too difficult. You only expand the last part of the regex with an alternative

```
"^Re(.*?):\s*(.*?)\s*(\was:.*\)|\(PGP Decrypted\))*$"
```

But now we have a look at the template. We would like to create a new subject. The macro we need is %SUBJECT Because we use it when we reply to a message and we want to have a proper subject line it should start with:

```
%SUBJECT="Re
```

Then we add the regex:

```
%SUBJECT="Re: %SETPATTREGEXP=""^Re(.*?):\s*(.*?)\s*(\was:.*\)|\(PGP
Decrypted\))*$""
```

[Note: the regex is wrapped due to layout reasons. All must be used as a single long line!]

We will apply it to the original subject %OFULLSUBJ and need the second subpattern. % SUBPATT="2"

```
%SUBJECT="Re: %SETPATTREGEXP=""^Re(.*?):\s*(.*?)\s*(\was:.*\)|\(PGP
Decrypted\))*$""%REGEXPBLINDMATCH=""%OFULLSUBJ""%SUBPATT=""2"""
```

[Note: the regex is wrapped due to layout reasons. All must be used as a single long line!]

Ok, that's it. Additional exercise: what if the subject does not have any 'Re' but 'AW', 'FWD' or anything else? Go, try to add further alternatives at the start of the regex.

Solution 2:

A positive lookahead assertion will help "`.*?(?=\@)`" The assertion will look for the @-character but won't include it in the match. Therefore, the template is easier to write:

```
%TO=""%TO='"%OFROMNAME at %-
%SETPATTREGEXP=_.*?(?=\@)_%-
%REGEXPMATCH=_%-
%OREPLYADDR_" <%OREPLYADDR>'
```

8. Final Conclusion

Now let's try to explain the example that was given in chapter 1.

```
%QUOTES="%SETPATTREGEXP=""(?is)(-----BEGIN PGP SIGNED.*?\n(Hash:.*?\n)?\s*)?
(.*?)(^(- --|--\n|-----BEGIN PGP SIGNATURE)|\z)""%REGEXPBLINDMATCH=""%text""%
SUBPATT=""3"""
```

It starts with %QUOTES=. The text that is matched with the following regex is to be used as quoted text.

"%SETPATTREGEXP="" defines the regex:

```
(?is)(-----BEGIN PGP SIGNED.*?\n(Hash:.*?\n)?\s*)?(.*?)(^(- --|--\n|-----BEGIN
PGP SIGNATURE)|\z)
```

[Note: the regex is wrapped due to layout reasons. All must be used as a single long line!]

You know already why there are doubled "-characters: it is to escape them so that they are not taken as part of another macro by mistake, (although you also know there are better ways of writing that too).

"`(?is)`" is the options setting: ignore case and assume the whole text as one single line, furthermore let the dot match newline characters.

```
"(-----BEGIN PGP SIGNED.*?\n(Hash:.*?\n)?\s*)?"
```

This opens the first subpattern. The regex says: find five hyphens followed by the string BEGIN PGP SIGNED. This may be followed by any character sequence or none at all (.*?). Due to the greediness of .* it is restricted by a question mark. Next is a following new line (\n).

The new line starts with the string 'Hash:', any character sequence and ends with a new line again. This is the second subpattern and it may appear once or never. Any number of whitespace characters may follow the second subpattern. Then the first subpattern is fully defined by the final parenthesis. Again this is followed by a question mark: that means that the first subpattern may appear only once or not at all.

These lines are created by PGP or GnuPG when a message is clear signed. The text is standard and therefore it is easy to define the regex. But the author of that macro combination not only wanted to use it on PGP-signed messages: he or she wanted to use it even on text that hasn't been touched by PGP and therefore do not have these lines.

```
"(.*?)"
```

This is the important third subpattern: the unmodified message text itself. The preceding regex was necessary to locate and isolate this subpattern. The regex just says: "Find anything, no matter what, but don't be greedy."

Now the alternation starts:

```
"(^(- --|--\n|-----BEGIN PGP SIGNATURE)|\z)"
```

Subpattern 4 starts and looks for a beginning of a line. Anything we now define in this subpattern has to be at the beginning of the line "(^". Then subpattern 5 follows:

```
"(- --|--\n|-----BEGIN PGP SIGNATURE)"
```

It consists of three alternatives:

```
"- --" resp. "--\n" or "-----BEGIN PGP SIGNATURE"
```

The first alternative is well known once you have seen a clear-signed PGP message : it is the modified signature separator that PGP uses with the extra hyphen and space as an indicator to show where it inserted its own lines. Quite unfortunate really, but we won't discuss it here. Just let's take it as is.

The second alternative is the original signature separator. That means that this will be found if the text had no contact with PGP. Actually, it's not quite right, because the proper cut mark is dash-dash-space-newline, so this regex should be:

```
"(^(- --|--\s\n|-----BEGIN PGP SIGNATURE)|\z)"
```

The third alternative is necessary to look f"(^(- --|--\s\n|-----BEGIN PGP SIGNATURE)|
\z)"or lines that contain the PGP-created hash (ok, ok, there is only a part of the hash, but this is a regex tutorial and not a PGP tutorial. If you need that one go to www.pro-privacy.de ;-)). This is the end of subpattern 5's definition.

The second alternative of subpattern 4 "\z)" searches for the end of the string as a counterpart to subpattern 5's search for the beginning of a line. Therefore there doesn't have to be a signature separator or a PGP-hash: The mail just has to end somewhere...

To be honest: the author looks for this funny ending of the mail only because of the fact that the proper text of the mail should be easily located and extracted. There is no further interest in these parts.

Now the next macro follows: %REGEXPBLINDMATCH=""%text"", which lets the machine apply the regex to the text.

The %SUBPATT=""3""" macro returns the proper part of the mail to the %QUOTES variable.

That's it.

A tutorial that is entirely written without direct feedback was something new to me: you don't notice when it gets too complicated or too academic. I tried to avoid both and I tried to concentrate on those elements of regular expressions that are most useful. I really hope I was successful and that it wasn't boring ;-)

The tutorial isn't a perfect and full description of regexian. If I wanted to offer that I could have copied J. Friedl's book into TB's help file. No, the tutorial was meant to give an idea, an initial help to get started. Like any other language you will only learn the vocabulary by doing and using it. If I was able to give you a hand to get started I'm content!

9. List of Elements of Regular Expressions and Macros

<table>
<tr><th>Character</th><th>Meaning</th></tr>
<tr><td>.</td><td>Any character except newline. With the DotAll option (?s) the dot matches newlines too</td></tr>
<tr><td>^</td><td>Matches beginning of a line</td></tr>
<tr><td>$</td><td>Matches end of line</td></tr>
<tr><td>...|...|...</td><td>Separates alternatives. First appearance in string is matched.</td></tr>
<tr><td>(...)</td><td>Is used for grouping search elements. The matched pattern is stored for further use.</td></tr>
<tr><td>[...]</td><td>Character class: characters in square brackets are alternatives. Ranges can be defined [a-p]. Some meta characters have different meaning here; [^...] negates the class.</td></tr>
<tr><td>\</td><td>Backslash: escapes the special meaning of meta characters: +?.*()^$[{|\</td></tr>
<tr><td>*</td><td>Finds multiple occurrences of the preceding character</td></tr>
<tr><td>+</td><td>Finds preceding character at least once</td></tr>
<tr><td>?</td><td>Finds any occurrence of the preceding character but not more than once.
OR
Switches off the greediness of quantifiers (.*?), only necessary minimum will be matched
OR
Introduces a modifier (?i)
OR
Introduces conditional regex (?(....)....|....) as well as assertions (?=...)</td></tr>
<tr><td>{x,y}</td><td>Finds preceding character x- to y- times ',y' is optional: {x} finds the element exactly x times. {x,} finds element at least x times.</td></tr>
<tr><td>\d</td><td>Represents digits</td></tr>
</table>

Character	Meaning
\D	Represents non-digits
\w	Represents alphanumeric characters and underscores.
\W	Represents non-\w characters
\s	Represents whitespace (tab, spaces usw.)
\S	Represents non-whitespace
\b	Represents word boundary
\B	Represents non-word boundary
\Z	Finds the end of a string
\z	Finds the physical end of a string
\A	Finds the beginning of a string
\1...\9	Backreferences. They point to any subpattern found in (...). More backreferences possible if more subpatterns are defined.
\nnn	Finds octal value of character nnn.
\xnn	Finds hexadecimal value of character nn
(?(condition pattern)yes pattern|No pattern)	Conditional regex: finds yes-pattern if condition-pattern is found in string, otherwise it tries to match the no-pattern. No-pattern is optional.
(?:…)	group will not be stored as a subpattern
…(?=…)	Positive Lookahead-Assertion. The pattern that precedes the parentheses will be searched if the pattern in the parentheses is found within the string that follows. The pattern in the parentheses is not part of the match.
…(?!…)	Negative Lookahead-Assertion. The pattern that precedes the parentheses will be searched if the pattern in the parentheses is not found within the string that follows. The pattern in the parentheses is not part of the match.
(?<=…)…	Positive Lookbehind-Assertion. The pattern that follows the parentheses will be searched if the pattern in the parentheses is found within the preceding string. The pattern in the parentheses is not part of the match.

Character	Meaning
(?<!…)…	Negative Lookbehind-Assertion. The pattern that follows the parentheses will be searched if the pattern in the parentheses is not found within the preceding string. The pattern in the parentheses is not part of the match.
(?Modifier)	Switch to set or unset modifiers for pattern matching. The modifier is set by adding it within the parentheses and is unset using a preceding hyphen. It is possible to specify several modificers in one go. (?i-s). i ignore case of character m string is taken as several lines, ^ and $ find any newline character within string s string is taken to be one long line, the dot matches newline characters within string. x enables comments and additional whitespaces that are not matched. To find spaces these literally have to be escaped by a backslash.

10. Examples for various regular expressions

Finally I would like to give some examples of regular expressions that could be helpful in everyday use and that provide examples of some of the syntax elements.

These examples have been taken from various Internet sources or purpose built.

Check mail-address

```
[\w-]+(?:\.[\w-]+)*@(?:[\w-]+\.)+[a-zA-Z]{2,7}
```

This Regex matches about 99% of all common email-addresses. There is no way to put together the definitive regex that will match all permutations (although Jeffrey Friedl published one: 6 kB long): it would end up much too complicated.

Check for valid IP-address

```
(25[0-5]|2[0-4][0-9]|[0-1]{1}[0-9]{2}|[1-9]{1}[0-9]{1}|[1-9])\.(25[0-5]|2[0-4]
[0-9]|[0-1]{1}[0-9]{2}|[1-9]{1}[0-9]{1}|[1-9]|0)\.(25[0-5]|2[0-4][0-9]|[0-1]
{1}[0-9]{2}|[1-9]{1}[0-9]{1}|[1-9]|0)\.(25[0-5]|2[0-4][0-9]|[0-1]{1}[0-9]{2}|
[1-9]{1}[0-9]{1}|[0-9])
```

The form of this regex should be familiar to you from when you solved one of the problems in the tutorial: check for a valid date. We divided the whole problem into smaller chunks that did the checking for us. Some combinations are not allowed (0.0.0.0 or any value bigger than 255) and you the IP address may be differently written (192.0.0.1 or 192.000.000.001): in both cases the regex will only match valid addresses.

Check for multiple matching recipients in an email address

```
((.*?)@.*?,\s*)(\2@.*?(,\s*)?){2,}
```

This check assumes that the recipients are listed in the header in the following way:

"`person@web.com, person@gmx.ch, person@weissnicht.de`"

All characters preceding the @-character are stored in subpattern 2 `((.*?)@.*?,\s*)`. The second part of the regex `(\2@.*?(,\s*)?)` uses the result of the first match and checks any following addresses using a back-reference to subpattern `\2`. This regex won't help if the recipients are written in the following way

```
"Person 1 <person@web.com>, Person 1 <person@gmx.ch>,   Person 1
<person@weissnicht.de>"
```

In this case you should use the more generic regex:

```
(((.*?)\s*<)?(.*?)@.*?>?,\s*)(((.*?)\s*<)?\4@.*?>?(,\s*)?){2,}
```

Beware: the back-reference has to be tweaked! The recipient within the email address is now matched using the fourth parentheses and is therefore backreference `\4`. In both regexes the number in the curly brackets represents how often the recipient will appear in the header.

Check for multiple matching domain names in email address

```
(.*?(@.*?),\s*)(.*?\2(,\s*)?){3,}
```

This is much more likely to happen than multiple matching names:

Albert.a@web.com, berta.b@web.com, charlie.c@web.com, dora.d@web.com

These will be matched with the given regex that, once again, uses a back-reference.

Conditional check for digits in the subject line

What this means is that only those mails that have several digits somewhere in the subject line are matched. This is quite often used to identify spam. But: an active eBay-seller or -buyer usually receives mails with a 10 digit article identification number in the subject line that would than be recognized as spam. Therefore we need a regex that uses the condition: “find several digits in the subject line unless preceded by the word 'article'”.

```
.*?(?<!article) \d{5,}
```

This is a look-behind assertion: it matches any 5-digit string unless preceded by 'article'.

Check for several consonants in a row

Some spammer use different mail addresses on different domains. The characteristic attribute is the fact that the address has quite a lot of consonants in series: fndghklxrrstU@example.com If you don't want to kill all mails from this domain on the server with the selective download filter you need another approach to this problem. One way is to define a character class that will allow all characters but no vowels:

```
(?i)[^aeiou]{5,}.*@
```

Let's assume that a more or less correct address does not have more than five consonants in a row then this regex should help to reduce spam. Attention: this may not work in all languages. Please check this for relevance to your own language.

11.Credits

I would like to thank those who helped convert my ideas into something readable and useful. My special thanks go to Marck who was very patient and who improved my translation. Thanks to (in alphabetical order):

Januk Aggarwal
Bert Bohla
Dirk Heiser
Hanja Nowicka
Thomas Martin
Peter Palmreuther
Marck D. Pearlstone
Stefan Peukert
Alfred Rübartsch
Andreas Rumpenhorst
Ingrid Spitzer
Carsten Thönges
Karin Uhlig
Arnd Wichmann
Thomas Wölk

Thank you to the Ooo-Team at www.openoffice.org for providing OpenOffice, which was used to write this tutorial (hey, you can use regex in OpenOffice for search and replace <g>).

The PDF-file was created with PDF995 and PDFedit from www.pdf995.com.