

~~maxbox~~

Regular Expression Report



Start with Regular Expressions Starter20 V2

1.1 From TRex to RegEx

Regular expressions are the main way many tools matches' patterns within strings. For example, finding pieces of text within a larger doc, or finding a restriction site within a larger sequence. This tutorial illustrates what a RegEx is and what you can do to find, match, compare or replace text of documents or code.

When I was a kid (just kidding) I was very impressed with a function in comparison with a RegEx. We are so amazed at how a function can do the job, because we sure never had the experience with a RegEx. And this is the function, it just strips all tags from a HTML page.

```
function StripTags2(const S: string): string;
var Len, i, APos: Integer;
begin
  Len:= Length(S);
  i:= 0;
  Result:= '';
  while (i <= Len) do begin
    Inc(i);
    APos:= ReadUntil(i, len, '<', s);
    Result:= Result + Copy(S, i, APos-i);
    i:= ReadUntil(APos+1, len, '>', s);
  end;
end;

Writeln(StripTags2('<p>This is text.<br/> This is line 2</p>'));
```

And now the same result with a RegEx:

```
Writeln(ReplaceRegExpr('<[>]*>',
  '<p>This is text.<br/> This is line 2</p>', '', True))
```

The Result will be in both cases: *This is text. This is line 2*

Impressive or not. We can argue that a function has the advantage to enlarge with objects and methods but the same goes also with a RegEx object (and there are many):

```

//replace StripTags with null
with TStRegex.Create(NIL) do begin
  //inputfile:= '<p>This is text.<br/> This is line 2</p>';
  inputfile:= exepath+'geomapX.txt';
  matchpattern.clear;
  matchpattern.add('<[^>]*>'); //find all tags and strip it!
  replacepattern.add('\z'); //Null expression!
  outputfile:= exepath+'geomapXoutreplace2.txt'
  Execute;
  Free
end;

```

As you will see regular expressions are composed of characters, character classes, meta-characters, quantifiers, and assertions.

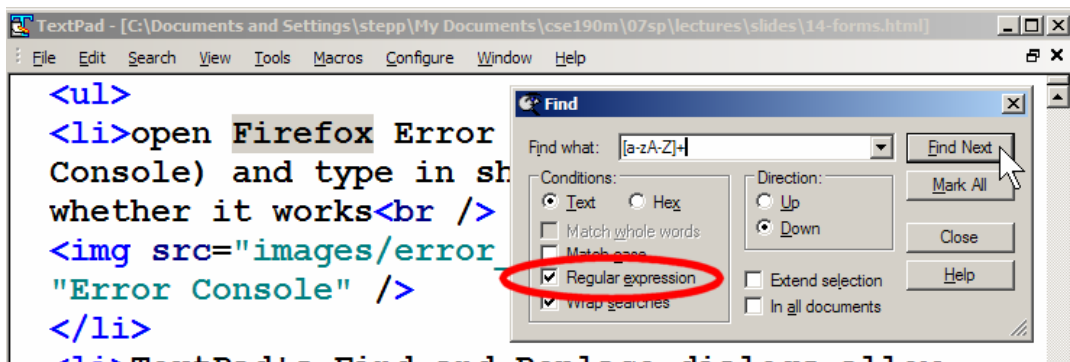
What's a Regular Expression?

A regular expression (RegEx): describes a search pattern of text typically made up from special characters called meta-characters:

- You can test whether a string matches an expression pattern
- You can use a RegEx to search/replace characters in a string
- It's very powerful, but a bit tough to read

Regular expressions occur in many places and masks, not alone in code or languages environments:

Text editors, shells or search tools like Grep¹ allow also RegEx in search/replace functions, like the following screen-shot figure out.



1: Find with RegEx

Let's jump to a history and the beginning of RegEx with Perl. Perl is a horribly flawed and very useful scripting language, based on UNIX shell scripting and C, that helped lead to many other better languages. Perl was and is also excellent for string/file/text processing because it built regular expressions directly into the language as a first-class data type.

👉 Many command-line shell Linux/Mac tools (ed, vi, grep, egrep, sed, awk) support Regular Expressions, for e.g. in Grep:

```

grep -e "[pP]hone.*206[0-9]{7}" contacts.txt
>> phone { 206-685-2181}

```

¹ Global Regular Expression Print / Parser

Grep is a tool that originated from the UNIX world during the 1970's. It can search through files and folders (directories in UNIX) and check which lines in those files match a given regular expression. Grep will output file-names and the line numbers or the actual lines that matched the regular expression.

1.2 RegEx out of the Box

As you already know the tool is split up into the toolbar across the top, the editor or code part in the centre and the output window at the bottom or the interface part on the right. Change that in the menu /View at our own style.



In maXbox you will execute the RegEx as a script, libraries and units are already built.



Before this starter code will work you will need to download maXbox from the website. It can be down-loaded from <http://www.softwareschule.ch/maxbox.htm> (you'll find the download maxbox3.zip on the top left of the page). Once the download has finished, unzip the file, making sure that you preserve the folder structure as it is. If you double-click `maxbox3.exe` the box opens a default demo program. Test it with F9 / F2 or press **Compile** and you should hear a sound. So far so good now we'll open the example:

```
309_regex_powertester4.txt
```

http://www.softwareschule.ch/examples/309_regex_powertester4.txt

Now let's take a look at the code of this first part project. Our first line is

```
01 program RegEx_Power_Tester_TReg4;
```

We name it, means the program's name is above.



This example requires two objects from the classes: `TRegExpr` and `TPerlRegEx` of `PerlRegEx` so the second one is from the well known `PCRE Lib`. `TPerlRegEx` is a Delphi VCL wrapper around the open source `PCRE` library, which implements `Perl-Compatible Regular Expressions`.

This version of `TPerlRegEx` is compatible with the `TPerlRegEx` class (`PCRE 7.9`) in the `RegularExpressionsCore` unit in Delphi XE. In fact, the unit in Delphi XE and `maXbox3` is derived from the version of `TPerlRegEx` that we are using now.

Let's do a first RegEx now. We want to check if a name is a valid Pascal name like a syntax checker does. We use straight forward a function in the box:

```
732 if ExecRegExpr('^ [a-zA-Z_] [a-zA-Z0-9_]*', 'pascal_name_kon')
    then writeln('pascal name valid') else writeln('pascal name invalid');
```

This is a useful global function:

```
function ExecRegExpr (const ARegExpr, AInputStr: string): boolean;
```

It is true if a string `AInputString` matches regular expression `ARegExpr` and it will raise an exception if syntax errors in `ARegExpr` are done.

Now let's analyse our first RegEx step by step `^[a-zA-Z_] [a-zA-Z0-9_]*`:

<code>^</code>	matches the beginning of a line; <code>\$</code> the end
<code>[a-z]</code>	matches all twenty six small characters from 'a' to 'z'
<code>[a-zA-Z_]</code>	matches any letter with underscore
<code>[a-zA-Z0-9_]</code>	matches any letter or digit with underscore
<code>.*</code> (a dot)	matches any character except <code>\n</code>
<code>. * *</code>	means 0 or more occurrences
<code>[]</code>	group characters into a character set;

A lot of rules for the beginning, and they look ugly for novices, but really they are very simple (well, usually simple ;)), handy and powerful tool too. You can validate e-mail addresses; extract phone numbers or ZIP codes from web-pages or documents, search for complex patterns in log files and all you can imagine! Rules (templates) can be changed without your program recompilation! This can be especially useful for user input validation in DBMS and web projects. Try the next one:

```
if ExecRegExpr('M[ae][iy]e?r.*[be]', 'Mairhuberu')
    then writeln('regex maierhuber true') else writeln('regex maierhuber false');
```


? Means 0 or 1 occurrences

Any item of a regular expression may be followed by another type of metacharacters – called iterators. Using this characters you can specify number of occurrences of previous characters so inside [], most modifier keys act as normal characters:

```
/what[.!*?]*-/ matches "what", "what.", "what!", "what?***!", ..
```

So a character class is a way of matching 1 character in the string being searched to any of a number of characters in the search pattern.

Character classes are defined using square brackets. Thus [135] matches any of 1, 3, or 5. A range of characters (based on ASCII order) can be used in a character class: [0-7] matches any digit between 0 and 7, and [a-z] matches any small (but not capital) letter.

 Note that the hyphen is being used as a metacharacter here. To match a literal hyphen in a character class, it needs to be the first character. So [-135] matches any of -, 1, 3, or 5. [-0-9] matches any digit or the hyphen.

What if we want to define a certain place? An assertion is a statement about the position of the match pattern within a string. The most common assertions are “^”, which signifies the beginning of a string, and “\$”, which signifies the end of the string.

For example search all empty or blank lines: Search empty lines: `^^$'`

This is how we can assert a valid port number with ^ and \$:

```
745 if ExecRegExpr('^(?:\d\d?\d?\d?\d?)$',':80009')
    then writeln('regex port true') else writeln('regex port false');
```

There are 3 main operators that use regular expressions:

1. **Matching** (which returns TRUE if a match is found and FALSE if no match is found.
2. **Substitution**, which substitutes one pattern of characters for another within a string
3. **Split**, which separates a string into a series of substrings

If you want to match a certain number of repeats of a group of characters, you can group the characters within parentheses. For example, `/(cat){3}/` matches 3 reps of “cat” in a row: “catcatcat”. However, `/cat{3}/` matches “ca” followed by 3 t’s: “cattt”.

And things go on. To negate or reject a character class, that is, to match any character EXCEPT what is in the class, use the caret ^ as the first symbol in the class. `[^0-9]` matches any character that isn’t a digit. `[^-0-9]` matches any character that isn’t a hyphen or a digit.

Now its time to reflect:

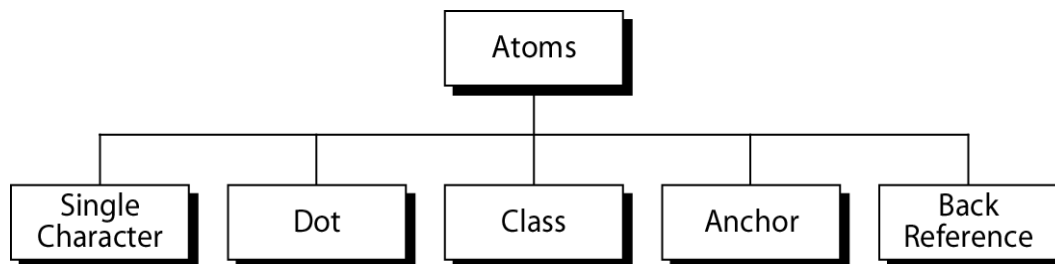
RE Metacharacter	Matches...
------------------	------------

^	beginning of line
\$	end of line
\char	Escape the meaning of <i>char</i> following it
[^]	One character <u>not</u> in the set
<	Beginning of word anchor
>	End of word anchor
() or \(\)	Tags matched characters to be used later (max = 9)
 or \ 	Or grouping
x\{m\}	Repetition of character x, m times (x,m = integer)
x\{m,\}	Repetition of character x, at least m times
x\{m,n\}	Repetition of character x between m and m times

2. Overview of Matches

You can specify a series of alternatives for a pattern using "|" to separate them, so that fee|fie|foe will match any of "fee", "fie", or "foe" in the target string (as would f(e|i|o)e). The first alternative includes everything from the last pattern delimiter ("('", "[", or the beginning of the pattern) up to the first "|", and the last alternative contains everything from the last "|" to the next pattern delimiter.

For this reason, it's common practice to include alternatives in parentheses, to minimize confusion about where they start and end.



Next a few examples to see the atoms:

```

rex:= '(no)+.*'; //Print all lines containing one or more consecutive
occurrences of the pattern "no".

rex:= '.*S(h|u).*'; //Print all lines containing the uppercase letter "S",
followed by either "h" or "u".

rex:= '.*\.[^0][^0].*'; //Print all lines ending with a period and exactly two
non-zero numbers.

rex:= '.*[0-9]{6}\. .*'; //all lines at least 6 consecutive numbers follow by a
period.
  
```

Next we want to see how the objects in the box work:

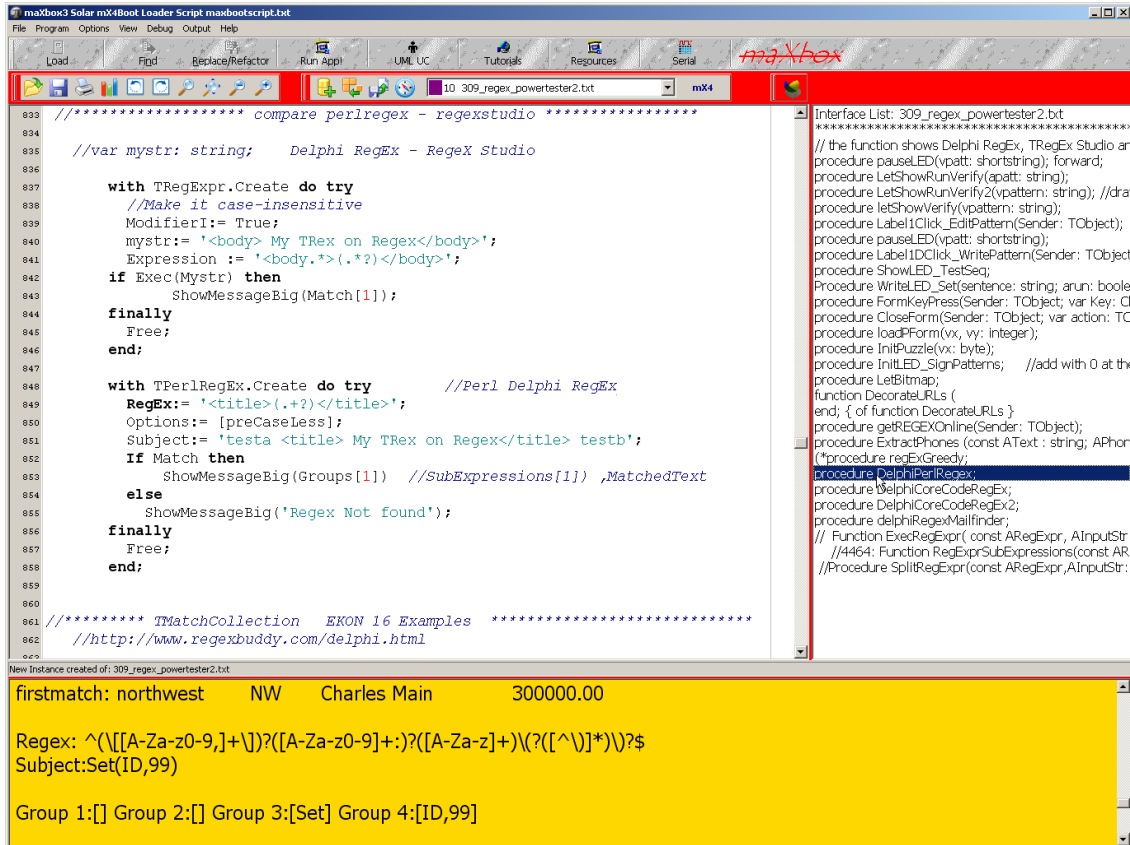
The static versions of the methods are provided for convenience, and should only be used for one off matches, if you are matching in a loop or repeating the same search often then you should create an 'instance' of the TRegEx record and use the non static methods.

The RegEx unit defines TRegEx and TMatch as records. That way you don't have to explicitly create and destroy them. Internally, TRegEx uses TPerlRegEx to do the heavy lifting.

TPerlRegEx is a class that needs to be created and destroyed like any other class. If you look at

the TRegEx source code, you'll notice that it uses an interface to destroy the TPerlRegEx instance when TRegEx goes out of scope. Interfaces are reference counted in Delphi, making them usable for automatic memory management.

☞ The XE interface to PCRE is a layer of units based on contributions from various people, the PCRE API header translations in RegularExpressionsAPI.pas



3: The two Classes in compare

Clients are seen in picture 3 as the two classes do the same task. After creating the object RegEx we set the options:

These are the most important options you can specify:

- **preCaseLess** Tries to match the regex without paying attention to case. If set, 'Bye' will match 'Bye', 'bye', 'BYE' and even 'byE', 'bYe', etc. Otherwise, only 'Bye' will be matched. Equivalent to Perl's /i modifier.
- **preMultiLine** The ^ (beginning of string) and \$ (ending of string) regex operators will also match right after and right before a newline in the Subject string. This effectively treats one string with multiple lines as multiple strings. Equivalent to Perl's /m modifier.
- **preSingleLine** Normally, dot (.) matches anything but a newline (\n). With preSingleLine, dot (.) will match anything, including newlines. This allows a multiline string to be regarded as a single entity. Equivalent to Perl's /s modifier.
- Note that preMultiLine and preSingleLine can be used together.
- **preExtended** Allow regex to contain extra whitespace, newlines and Perl-style comments, all of which will be filtered out. This is sometimes called "free-spacing mode".
- **preAnchored** Allows the RegEx to match only at the start of the subject or right after the previous match.
- **preUngreedy** Repeat operators (?, *, +, {num,num}) are greedy by default, i.e. they try to match as many characters as possible. Set preUngreedy to use

ungreedy repeat operators by default, i.e. they try to match as few characters as possible.



Greedy is a strange operator or option. A slight explanation about "greediness".

"Greedy" takes as many as possible; "non-greedy" takes as few as possible. For example, 'b+' and 'b*' applied to string 'abbbbc' return 'bbbb', 'b+?' returns 'b', 'b*?' returns an empty string, 'b{2,3}?' returns 'bb', 'b{2,3}' returns 'bbb'.

The regular expression engine does "greedy" matching by default!

A typical RegEx client session looks like this:

```
848 with TPerlRegEx.Create do try           //Perl Delphi RegEx
849     RegEx:= '<title>(.*?)</title>';
850     Options:= [preCaseLess];
851     Subject:= 'testa <title> My TRex on Regex</title> testb';
852     If Match then
853         ShowMessageBig(Groups[1]) //SubExpressions[1] ,MatchedText
854     else
855         ShowMessageBig('Regex Not found');
856 finally
857     Free;
858 end;
```

Subject is the RegEx and the string on which Match will try to match RegEx. Match attempts to match the regular expression specified in the RegEx property on the string specified in the Subject property. If Compile has not yet been called, Match will do so for you.

Call MatchAgain to attempt to match the RegEx on the remainder of the subject string after a successful call to Match.



Compile: Before it can be used, the regular expression needs to be compiled. Match will call Compile automatically if you did not do so. If the regular expression will be applied in time-critical code, you may wish to compile it during your application's initialization. You may also want to call Study to further optimize the execution of the RegEx.

Let's have a look at the RegEx itself and the magic behind:

```
'<title>(.*?)</title>'
```

What's about this <title>, it must be a global identifier to find or still exists.

```
+? one or more ("non-greedy"), similar to {1,}?
```

It captures everything between the first <title> and the first </title> that follows. This is usually what you want to do with large sequences in a group.



Greedy names: Greedy matching can cause problems with the use of quantifiers. Imagine that you have a long DNA sequence and you try to match /ATG(.*?)TAG/. The ".*" matches 0 or more of any character. Greedy matching causes this to take the entire sequence between the first ATG and the last TAG. This could be a very long matched sequence.



Note that Regular expressions don't work very well with nested delimiters or other tree-like data structures, such as are found in an HTML table or an XML document. We will discuss alternatives later in a course.



So far we have learned little about RegEx and the past with a TRex eating words as a book output to us;-). Now it's time to run your program at first with F9 (if you haven't done yet) and learn something about the 309_regex_powertester2.txt with many code snippets to explore.

One of them is a song finder in the appendix to get a song list from an mp3 file and play them!



4: Mastering

There are plenty more regular expression tricks and operations, which can be found in Programming Perl, or, for the truly devoted, *Mastering Regular Expressions*. Next we enter part two of the insider information about the implementation.



5: Enter a RegEx building

1.3 RegEx in Delphi and maXbox

The `TPerlRegEx` class aims at providing any Delphi, Java or C++Builder developer with the same, powerful regular expression capabilities provided by the Perl programming language community, created by Larry Wall.

It is implemented as a wrapper around the open source `PCRE` library.

The regular expression engine in Delphi XE is PCRE (Perl Compatible Regular Expression). It's a fast and compliant (with generally accepted RegEx syntax) engine which has been around for many years. Users of earlier versions of Delphi can use it with `TPerlRegEx`, a Delphi class wrapper around it.

`TRegEx` is a record for convenience with a bunch of methods and static class methods for matching with regular expressions.

I've always used the `RegularExpressionsCore` unit rather than the higher level stuff because the core unit is compatible with the unit that Jan Goyvaerts has provided for free for years. That was my introduction to regular expressions. So I forgot about the other unit. I guess there's either a bug or it just doesn't work the way one might expect.

For new code written in Delphi XE, you should definitely use the `RegEx` unit that is part of Delphi rather than one of the many 3rd party units that may be available. But if you're dealing with UTF-8

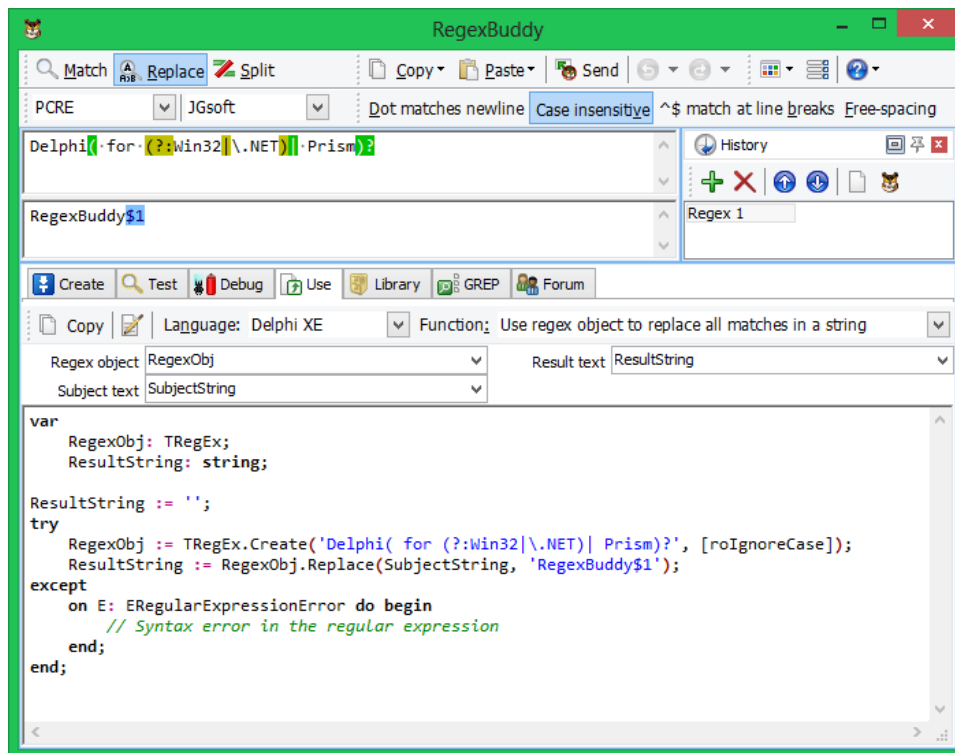
data, use the `RegularExpressionsCore` unit to avoid needless UTF-8 to UTF-16 to UTF-8 conversions.

☞ The procedure `Study` procedure `Study`;

Allows studying the RegEx. Studying takes time, but will make the execution of the RegEx a lot faster. Call `study` if you will be using the same RegEx many times. `Study` will also call `Compile` if this had not yet been done.

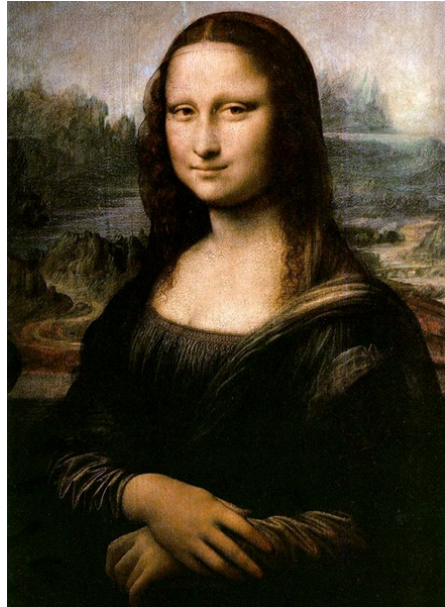
Depending on what the user entered in `Edit1` and `Memo1`, RegEx might end up being a pretty complicated regular expression that will be applied to the memo text a great many times. This makes it worthwhile to spend a little extra time studying the regular expression (later on more).

By the way there's another tool: Compose and analyze RegEx patterns with RegxBuddy's easy-to-grasp RegEx blocks and intuitive RegEx tree, instead of or in combination with the traditional RegEx syntax. Developed by the author of the website <http://www.regular-expressions.info/>, RegxBuddy makes learning and using regular expressions easier than ever.



6: The RegxBuddy and the GUI

Conclusion: A regular expression (RegEx or regexp for short) is a special text string for describing a search pattern. You can think of regular expressions as wildcards on steroids. You are probably familiar with wildcard notations such as `*.txt` to find all text files in a file manager. The RegEx equivalent is `.*\s.txt$`.




7: The Secret behind this Regular Expression!?

Study method example:

Depending on what the user entered in `Edit1` and `Memo1`, `RegEx` might end up being a pretty complicated regular expression that will be applied to the memo text a great many times. This makes it worthwhile to spend a little extra time studying the regular expression.

```
31 with PerlRegEx1 do begin
    RegEx:= Edit1.Text;
    Study;
    Subject:= Memo1.Lines.Text;
    Replacement:= Edit2.Text;
    ReplaceAll;
    Memo1.Lines.Text:= Subject;
end;
```



 Try reformat phone numbers from 206-685-2181 format to (206) 685.2181 format to get data back:

You can use back-references when replacing text. Text "captured" in () is given an internal number; use `\number` to refer to it elsewhere in the pattern `\0` is the overall pattern, `\1` is the first parenthetical capture, `\2` the second, ...

Example: "A" surrounded by same character: `/(.)A\1/`

Example: to swap a last name with a first name:

```
var name = "Durdan, Tyler";
    name = name.replace(/(\w+),\s+(\w+)/, "$2 $1");
    // "Tyler Durdan"
```

maxbox



Time of day: For example. 11:30. [01][0-9]:[0-5][0-9] won't work well, because it would allow such impossible times as 19:00 and 00:30. A more complicated construction works better: (1[012] | [1-9]) : [0-5][0-9]. That is, a 1 followed by 0, 1, or 2, OR any digit 1-9.

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769 770 771 772 773 774 775 776 777 778 779 780 781 782 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831 832 833 834 835 836 837 838 839 840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858 859 860 861 862 863 864 865 866 867 868 869 870 871 872 873 874 875 876 877 878 879 880 881 882 883 884 885 886 887 888 889 890 891 892 893 894 895 896 897 898 899 900 901 902 903 904 905 906 907 908 909 910 911 912 913 914 915 916 917 918 919 920 921 922 923 924 925 926 927 928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981 982 983 984 985 986 987 988 989 990 991 992 993 994 995 996 997 998 999 1000
```

Feedback @ max@kleiner.com

Literature:
Kleiner et al., Patterns konkret, 2003, Software & Support

Links of maXbox and RegEx EKON Slide Show:

- <http://www.softwareschule.ch/maxbox.htm>
- http://www.softwareschule.ch/download/A_Regex_EKON16.pdf
- <http://www.regular-expressions.info/>
- http://regexpstudio.com/tregexpr/help/RegExp_Syntax.html
- <http://sourceforge.net/projects/maxbox>
- <http://sourceforge.net/apps/mediawiki/maxbox/>
- <http://sourceforge.net/projects/delphiwebstart>

1.4 Appendix

EXAMPLE: Mail Finder

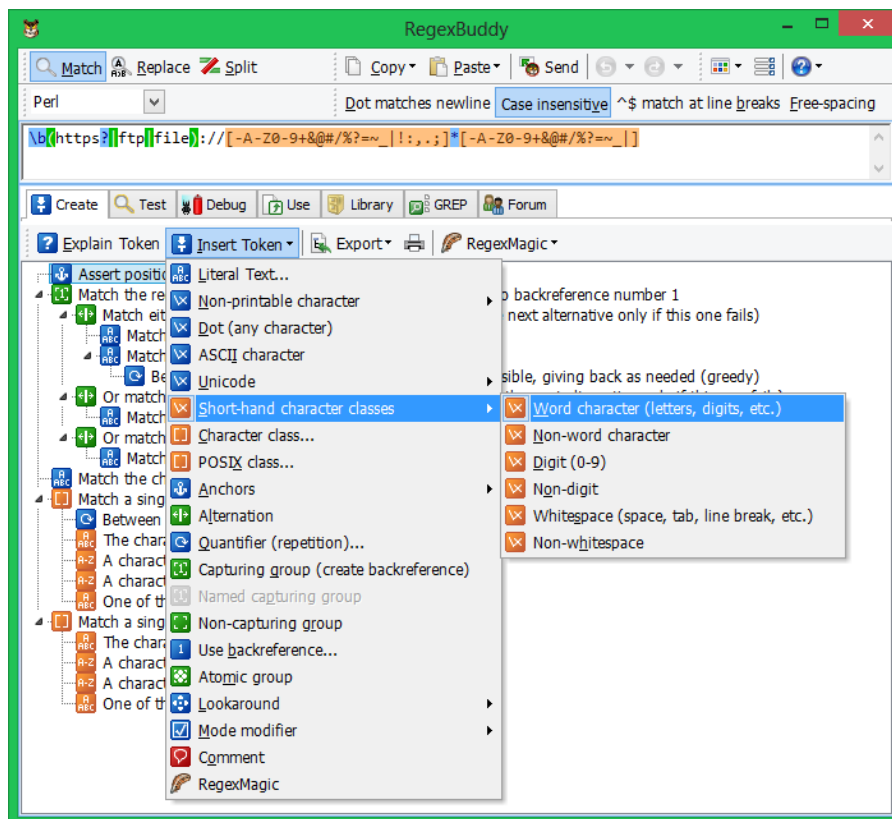
```
procedure delphiRegexMailfinder;
begin
  // Initialize a test string to include some email addresses. This would
  normally be your eMail.
  TestString:= '<one@server.domain.xy>, another@otherserver.xyz';
  PR:= TPerlRegex.Create;
  try
    PR.RegEx:= '\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b';
    PR.Options:= PR.Options + [preCaseLess];
    PR.Compile;
    PR.Subject:= TestString; // <-- tell PR where to look for matches
    if PR.Match then begin
      WriteLn(PR.MatchedText); // Extract first address
      while PR.MatchAgain do
        WriteLn(PR.MatchedText); // Extract subsequent addresses
      end;
    finally
      PR.Free;
    end;
  //ReadLn;
end;
```

EXAMPLE: Songfinder

```
with TRegExpr.Create do try
  gstr:= 'Deep Purple';
  modifierS:= false; //non greedy
  Expression:= '#EXTINF:\d{3},'+gstr+' - ([^\n].*)';
  if Exec(fstr) then
    Repeat
      writeln(Format ('Songs of ' +gstr+': %s', [Match[1]]));
      (*if AnsiCompareText(Match[1], 'Woman') > 0 then begin
        closeMP3;
        PlayMP3('\..\EKON_13_14_15\EKON16\06_Woman_From_Tokyo.mp3')
      ;
      end;*)
    Until Not ExecNext;
  finally Free;
end;
```

```
//***** Code Finished*****
```

1.5 Appendix RegExBuddy in Action



1.6 Appendix String RegEx methods

<code>.match(<i>regexp</i>)</code>	returns first match for this string against the given regular expression; if global /g flag is used, returns array of all matches
<code>.replace(<i>regexp</i>, <i>text</i>)</code>	replaces first occurrence of the regular expression with the given text; if global /g flag is used, replaces all occurrences
<code>.search(<i>regexp</i>)</code>	returns first index where the given regular expression occurs
<code>.split(<i>delimiter</i>[, <i>limit</i>])</code>	breaks apart a string into an array of strings using the given regular as the delimiter; returns the array of tokens