TSO Rexx API for PCRE (Perl Compatible Regular Expressions)

Copyright 2020 © Ze'ev Atlas and John Gateley

Table of Contents

Disclaimer: The PCRE for z/OS library port works on EBCDIC code pages only. The API may work on some ASCII code pages providing that the IBM supplied [iconv] facility could properly convert that code page to IBM-1047 and back, but this has never been tested.

Neither the library port, nor the API could ever handle wide characters of any sort. Neither any UTF nor any DBCS or MBCS is supported. No such support is ever planned.

The original PCRE library (not this port or API) runs on Linux and UNIX and supports UTF in all its glory. I have never tried that on the mainframe as I have no access to Linux on the mainframe.

What are Regular Expressions REGEX) and why Using Them

A Regular Expression is a way to define text patterns when performing searches on text strings and for substituting the search result with another pattern. It is currently the most popular method for automating searches and replacements in text strings. Its capabilities surpass the capability of the Rexx 'parse' function and other related text functions by several orders of magnitude.

Let's look at a few, rather simple Regular Expression examples using Perl notation, which is the most popular and the notation used by the PCRE library:

The Regular expression [/cat/i] finds the first appearance of the string 'cat' in the subject string, regardless of case. This is rather a dangerous way to do it if you really want to find the first instance of ‘cat’, because it will find the 'Cat' in "Catch"; searching the string "Catch the cat please". For a more accurate search, limit the search to the word 'cat' using the expression `[/\bcat\b/i]`. And, if the search requires only the string ‘cat’ when followed by ‘dog’ but without the ‘dog’ characters, enhance the expression as [/\bcat(?=\sdog)/i].

Consider the Regular Expression [/\b(?:[0-6]\d:){2}[0-6]\d\b/g]. This pattern finds all appearances of a time string (i.e. hh:mm:ss) in a long string containing several instances in one pass. This example contains additional strange notations and modifiers, so let’s explain some of them:

\b – limits the result to a word border back or front
\s – includes white space in the result
\d – includes a digit in the result
[0-6] – includes any digit in the range of 0 to 6 in the result
{2} – previous group, exactly two times

Groups are formed by parentheses. The [(?:xxx)] notation means that it is non-capturing group, and the default, simple parentheses (xxx) is a capturing group. So we have a means to capture portions of the matching pattern. And if we have a means to capture these groups selectively we could do two major things:

1. To present a vector of captured groups
2. To substitute these groups, or the whole pattern with something else.

We can even name the groups if we do not like to count. (There would be an example later.)

Regular Expressions Usage

I limit my examples, as this session is not about regular expressions but about their usage in TSO Rexx. You can see that for the uninitiated, regular expressions may be quite cryptic and seems unsavory. Yet, as I've mentioned, it is the most popular way to do it. To prove this point, I looked at the TIOBE Index July 2020 (https://www.tiobe.com/tiobe-index/) top few popular languages and their Regular Expression usage:

1. C - Native Posix functions, PCRE, other 3rd party libraries
2. Java - native class 'java.util.regex'
3. Python - native class 'import re'
4. C++ - native classes '<regex>'
5. C# - native class 'public class Regex'
6. Visual Basic - native class 'Imports System.Text.RegularExpressions'
7. JavaScript - native operators, 'RegExp' object
8. R - Native Posix functions, PCRE
9. PHP - PCRE

... Skipped some less important languages…

14. Perl - native operators
15. MATLAB - native functions
16. Ruby - native operators

These are the first top 16 languages. For a perspective COBOL is number 28, PL/I is number 137 and Rexx does not even make it to the first 150.

Regular Expression (REGEX) functionality for Rexx previously implemented:

Now, John and I are not the first people to introduced Regular Expressions into Rexx.

- OORexx comes with RxRegExp - An external function package providing search capabilities with Regular Expressions.
- Regina Rexx - you may download RexxUtil for Regina which includes RexxRE Regular Expression Library that provides POSIX regular expressions for Rexx.
- NetRexx may use the Java classes (see Java above)

I will not weigh here about the relative qualities of Regular Expressions vs. using ‘parse’ as this is a pointless discussion. Instead, I would say, use whatever works for you but you should have the opportunity to use any or both.

The PCRE Library

The PCRE (PCRE.org) library has been developed (in C) by Phil Hazel of Oxford University, in order to provide a Perl Compatible Regular Expression library for languages and other products that do not have this capability.

The Perl compatibility is important due to the Perl flavor of Regular Expressions becoming the standard; surpassing the Posix compliant version. Other languages usually thrive for Perl compatibility. Both the R language and PHP for example use the PCRE library as well as many other languages and products.

I ported PCRE to Classic z/OS (EBCDIC, JCL, LE API, etc.) on 2012 and I keep it current. It is available on CBTTAPE.org file 939. It is now PCRE2 version 10.35. Older versions of PCRE are not supported. For compatibility reasons, I use only the IBM C/C++ compiler.

The original PCRE library is interpretative and handles 8 bit ASCII, UTF-8, UTF-16 and UTF-32. It also has a JIT (Just In Time compiler) counterpart that can handle Intel x86, SPARC, MIPS, ARM and PPC hardware.

My aim with my port was to address EBCDIC systems and the poor souls that were stuck with ancient COBOL (and PL/I) programs on Classic z/OS. At the time we did not develop JIT for the IBM z series as we considered it to be too complicated, requires knowledge that we did not possess (about the hardware) and of too little demand. Thus, the current port is interpretative and handles 8 bit EBCDIC in interpretive mode only.

On 8/21/2020, IBM contacted the PCRE development team and published this contribution that is aimed to: "Enable PCRE2 JIT on Linux on IBM-z (s390x-linux) and optimize to achieve equivalent speedup over non-JIT code as x86_64. Goal is full functionality and passing test suite with JIT enabled. An unfinished port of PCRE2 JIT to s390x exists in the Linux-on-IBM-z Github account and can be used as a starting point. IBM will contribute the code as necessary."

In other words, IBM is endorsing the PCRE2 library on IBMz systems. I will probably incorporate that when it becomes available.

The TSO Rexx API for PCRE

After completing the port and developing a full API for LE languages (COBOL), the next natural step was to develop an API for TSO Rexx. With John Gateley's help we have accomplished that goal.
Our objectives:

1. Provide the main functionality of the package, i.e. compile of Regular Expression, search and substitute. The COBOL/LE API supplies the full functionality of the PCRE library, but I felt that Rexx users would probably not need the more esoteric functions that look for optimization, context related and other fancy stuff. I could change this design goal in light of future popular demand.
2. Provide results in native Rexx way, Rexx stems and variables.
3. Seamlessly handling the difficulties around managing EBCDIC code pages.

The Rexx API was developed using a mix of C and Assembler routines. In order to maintain z/OS compatibility I strictly use IBM C/C++ compiler and IBM supplied C runtime library. The Assembler code relies on the Concept14 macros (CBTTAPE.org file 953.)

The PCRE library and the API are licensed in BSD style Open Source license. This is a very permissive license that basically means, use it however you wish, just mention our names. I avoided the restrictive GPL license because of that the original library is BSD licensed and I also felt that using a restrictive license would prevent any usage of the library in the corporate world.

The Rexx API in Detail

The API uses the following functions for the C environment

- EDCXHOTL: Create C environment with library functions
- EDCXHOTU: Call a C routine
- EDCXHOTT: Terminate C environment

This method is based on the article "Calling C functions from Assembler - revisited" by A Rudd printed in issue 208 of Xephon magazine of January 2004,
http://www.cbttape.org/xephon/xephonm/mvs0401.pdf .
The CEEPIPI function may also work, and is recommended by IBM. We may look at it in subsequent releases.

We considered two distinct scenarios for using PCRE via REXX:

1. Parsing a value, such as an argument to the REXX program, or a single record in a control file.
2. Performing regular expression functions on every record from a file.

To easily enable these two scenarios independent interfaces between REXX and PCRE were developed.

1. RXPCRE2: An interface between the Rexx language on z/OS and the PCRE2 - regex processing library on same platform for iterative calls.
2. REXXPCRE: An interface for non-iterative calls.

- REXXPCRE: A Rexx function for single execution of the RegEx. This combine connect, compile, match, release and disconnect in a single call. It does not support substitute processing.
- RXPCRE2: A Rexx function for multiple executes of the RegEx. This program should be called from a Rexx program to handle Perl compatible regular expressions. It gets the Rexx arguments, including the desired command (function) and passes them to RXPCRE2A, a separate load module that also obtains the C code. On first call it loads RXPCRE2A into memory and returns the load address as a handle, to the calling Rexx function so that subsequent calls to this module can use the same instance that was previously loaded.

Returned results from this function depend on the command given.
Any error messages from the program are printed using a call to IRXSAY, the 'SAY' callable function.
Note: see a detailed list of available commands and their arguments below.

The implemented functionality depends on the following modules:

- RXPCRE2A: used by RXPCRE2 and contains the calls to all the C code and should only be called by RXPCRE2 as it uses the Rexx environment that exists in that module.
- PROCSTEM: manipulates Rexx stem variables on behalf of the calling program. Macro PRGSTEM should be used in the calling program to define the interface storage. This module also makes use of the Rexx environment.
- STRINGIT: used by the STRING macro - acts like COBOL STRING.
- TRIMIT: used by the TRIM macro to remove excess spaces.
- RXPCRE2O: an optional supporting function was added to produce option words. Used to create option integers for processing in RXPCRE2.

The API is using the PCRE2 C library, additional C functions that were written to help with z/OS and EBCDIC code pages and supplied standard C runtime library. Notably, [iconv], [iconv_open] and [iconv_close] are used to handle the various EBCDIC code pages.

Note on EBCDIC Code Pages:

Various EBCDIC code pages place the dollar symbol in different code points. In IBM-037 the dollar is x'5B' but in IBM-285 (UK) this is the currency symbol for GBP and the dollar is at x'4A' which is the cent symbol in IBM-037. Confusing!
Internally PCRE2 uses IBM-1047 so to use the dollar symbol in the regex we pass a fifth argument which is the code page name we are actually using. This causes PCRE to convert the regex internally from IBM-285 to IBM-1047 before it is compiled; similarly the string is converted before the match is done. And if needed it is converted back to the original code page. Thus, once you supply the desired code page, the rest is done seamlessly by the API.

NOTE - supplying code page name is required if you use a code page in your terminal emulator which is NOT the same as the default local code page. Normal processing by PCRE is to convert the REGEX and subject string from the default local code page to IBM-1047. IBM-1047 is used by the mainframe C compiler.

Detailed usage of REXXPCRE

REXXPCRE is a one stop function for all functions provided by the API. It is meant for a single use of a Regular Expression.

INPUT ARGUMENTS:

1. A string containing the regular expression.
2. The subject string to be worked on.
3. A string that contains STEM name which will be created and populated with the output
4. Optional OPTION or any combination thereof
   'g' - match all
   'i' - ignore case
   'x' - exclude white space
5. Optional string containing the code-page - e.g. 'IBM-285'. If omitted the default local page is used
6. 'debug' if debugging required. Needs PCREDUMP DD in the JCL.

OUTPUT

```
        if  successful
           RC=1   match  - stem contains output
           RC=0   no match - stem.0 will be '0'
```

```
else
   RC='error message text'
endif
```

STEM variables structure

stem.0	count of stem variables
stem_STRING.?	matched substrings
stem_POS.?	position and length of matches e.g. 3,4
stem_NAME.?	named substrings, ' ' if none

DEBUGGING can be done by specifying 'debug' as argument #6.

You will also need to allocate the DD name PCREDUMP with DISP=MOD if more than one call to REXXPCRE because each call is a separate unit of work which includes opening and closing the dataset.

Detailed usage of RXPCRE2

RXPCRE2 is a Rexx function and RXPCRE2A is its helper module.
On the 'connect' call, RXPCRE2A is loaded into memory of the program is returned, this enables the module instance to be used on subsequent calls.
The 'disconnect' function terminates the C environment and deletes RXPCRE2A from memory.

There are 6 available commands to the function

1. 'CONNECT' - connect to the C environment
   Establish the C environment with or without debug. If OK returns an eight byte environment handle which should be used in subsequent calls to the function. This must be called first.
   Parameters
   - Command - string contains ‘connect’
   - Environment handle - string contains the name of the environment handle variable
   - optional name of code page
   - optional string contains ‘debug’

   Examples:
   ```
   r_c    = RXPCRE2('connect','pcre_env')
   r_c    = RXPCRE2('connect','pcre_env','debug')
   r_c    = RXPCRE2('connect','pcre_env','IBM-285')
   r_c    = RXPCRE2('connect','pcre_env','IBM-285','debug')
   ```
   Returns 0 if OK
   8 if failed - an error message will be printed using IRXSAY

2. ‘COMPILE’ (alias ‘COMP’)

Compile a PCRE regular expression. This must be called first for every individual regular expression. If OK, returns nineteen bytes compile handle which should be used in subsequent calls related to that RegEx.
Parameters

- Command – string contains ‘compile’ or ‘comp’
- Environment handle variable that was created in the ‘connect’ call
- A string that contains the regular expression
- A string that contains the name of the compile handle variable
- Optional – possible option
- Optional option word

Option or any combination thereof can be specified

g' repeat search
i' ignore case
x' exclude white space

Note that multiple regular expressions can be compiled and then executed in turn using different compile handles.
Example:

```
reg_ex  = "(?<char>A)\g<char>"
 r_c     = RXPCRE2('compile',pcre_env,reg_ex,'pcre_comp','gi', op_wrdc)
```

Returns 0 if OK
8 if failed - an error message will be printed using IRXSAY

3. ‘MATCH’ (aliases ‘EXECUTE’ or ‘EXEC’)
Process the compiled expression on the subject string, repeat this call for every new instance of the subject string as required. Returns a stem with match results.
Parameters
    - Command - string contains ‘compile’ or ‘comp’
    - Environment handle variable that was created in the ‘connect’ call
    - Compile handle variable that was created in the ‘compile’ call
    - A string that contains the subject string
    - A string that contains the stem name for the match results
    - Optional option word

STEM variables structure is the same as in REXXPCRE

stem.0	count of stem variables
stem_STRING.?	matched substrings
stem_POS.?	position and length of matches e.g. 3,4
stem_NAME.?	named substrings, ' ' if none

Example:

```
the_str   = "AN_AARDWARK_JAKE_AND_AARDWARK_JACK"
my_stem   = "WANG"
r_c = RXPCRE2('match',pcre_env,pcre_comp,the_str,my_stem,op_wrd1)
```

Returns 0 if no match – stem.0 contains zero.
    1 matched - stem.string.? stem.POS.? and stem.NAME.? contain results
    8 if failed - an error message will be printed using IRXSAY

4. SUBSTITUTE – (alias 'SUBS')
   Similar to MATCH, but requires an additional input containing the substitute string or pattern. Process the compiled RegEx and on the subject string and substitute matches with replacement that could be a string or a pattern. Also returns a variable that contains the output string
   Parameters
   - Command - string contains 'substitute' or 'subs'
   - Environment handle variable that was created in the 'connect' call
   - Compile handle variable that was created in the 'compile' call
   - A string that contains the subject string
   - A string that contains the stem name for the match results
   - A string that contains the name of the Rexx variable to contain the result string after substitution
   - A string that contains the substitution string or pattern
   - Optional option word

   Example:

```
reg_ex   = "cat|dog"
/*this would be the input to the 'compile' that is represented here by the pcre_comp handle*/

the_str  = "the dog sat on the cat's dog"
subs_str = "horse"
my_var   = "WANG"
r_c = RXPCRE2('substitute',pcre_env,pcre_comp,the_str, my_var,subs_str,op_wrd1)
```

   Output world be: = "the horse sat on the horse's horse" of option 'g' was chosen.

   returns 0 no match - Rexx variable not set
       1 match - Rexx variable contains output string
       8 an error message will be writen using IRXSAY

5. RELEASE
   Release the compile storage
   Parameters
   - Command – string contains 'release'
   - Environment handle variable that was created in the 'connect' call
   - Compile handle variable that was created in the 'compile' call

   Example:

```
re_lease  = RXPCRE2('release',pcre_env,pcre_comp)
```

Returns 0 if successfull.
    8 if failed - an error message will be printed using IRXSAY

6. DISCONNECT
   Terminate the C environment.
   Parameters
   - Command – string contains ‘release’
   - Environment handle variable that was created in the ‘connect’ call

   Example:
   ```
   r_c     = RXPCRE2('disconnect',pcre_env)
   ```

   Returns 0 if successfull.
       8 if failed - an error message will be writen using IRXSAY

NOTE: if any call fails then all storage and compile handles will be released and the C environment will be terminated. Do not attempt to continue after this.

Additional Optional Word and the RXPCRE2O function

Advanced users may use “named option bits”, found in the PCRE documentation. They may be used to signal slight behavior changes. These bits, depend on which one, could be applied in compile time, match time and / or substitute time. You may see a list of those bit names and where are they supported (not all are supported in the PCRE port) in the appendix.

RXPCRE2O is an optional supporting function was added to produce option words out of those bit names. It is used to create an option integer for processing in RXPCRE2.

There is one call to the function

INPUT ARGUMENTS:
1. The name of a stem variable which contains the options. See structure below
2. the name of a Rexx variable which will contain the resulting option bit values. This will be 8 bytes of hexadecimal display

Example:
```
op_t.1 = 'PCRE2_DOLLAR_ENDONLY'
op_t.2 = 'PCRE2_DOTALL'
op_t.3 = 'PCRE2_DUPNAMES'
op_t.0 = '3'
r_c     = RXPCRE2O('OP_T','VAR_NAM')
```

```
Returns 0 if OK
        8 if failed - an error message will be printed using IRXSAY
```

Runtime notes

When running in batch the LOAD macro will try to find the requested module/program object in the STEPLIB or JOBLIB concatenation which will contain the PDSE library containing RXPCRE2A.
In ISPF the library would be concatenated to ISPLLIB which will allow RXPCRE2 to be loaded, however, when this tries to load RXPCRE2A the STEPLIB would be used and would fail. It should be possible to put the program library in the ISPLLIB concatenation and amend all references to PCRELIB to ISPLLIB but this will not work when ISPLLIB is modified using LIBDEF as the library is not added to the actual ISPLLIB but to another DDNAME which is logically concatenated by ISPF.
For the above reasons PCRELIB is used. If it is not present in the task IO table the program will not attempt to use it.

```
/* allow RXPCRE2 to be loaded by REXX */
"ISPEXEC LIBDEF ISPLLIB DATASET ID ('SDJRG.LOADLIB.POBJ')"
if rc /= 0 then do
  say 'allocation to ispllib failed.'
  exit
end
/* allow RXPCRE2A to be loaded by RXPCRE2A */
"ALLOC FI(PCRELIB)  DA('?????.LOADLIB.POBJ') SHR"
if rc /= 0 then do
  say 'allocation to PCRELIB failed.'
  exit
end

...

"FREE FILE(PCRELIB)"
"ISPEXEC LIBDEF ISPLLIB "
```

Appendix:

Sample code:

RXPCRE2B is a sample program that demonstrates usage of both REXXPCRE and RXPCRE2 for matching

```
/*REXX*/

  the_str1 = "The quick brown fox jumps over the lazy dog."
  the_str1 = the_str1||"The quick brown fox jumps over the"
  the_str2 = "The brown quick fox over the lazy dog jumps."
  the_str2 = the_str2||"The brown quick fox over the jumps"
  the_str3 = "The slow brown fox walks past the lazy dog."
  the_str4 = "The slow brown fox walks quickly past the lazy dog."

  reg_ex_9   = "(?<char>A)\g<char>"    /* ||'00'x */
  the_str9   = "AN_AARDWARK_JAKE_AND_a AARDWARK_JACK"
  the_str9a  = "ANnn_AARDWARK_JAKE_AND_a lower AARDWARK_JACK"
  opt_ion_9  = "gix"
  reg_ex    = "(quick|jump)"            /* ||'00'x */
  my_stem = "WANG"
  opt_ion = "g"

/*
  say '  '

  say ' first the old version +++++++++++++++++++++++++++ '

  say '  '


  say reg_ex
  say the_str1

  re_sponse = REXXPCRE(reg_ex,the_str1,my_stem,opt_ion,,'debug')

  select
     when re_sponse = 1 then do
        say 'wang.0  is  ' wang.0
        do lo_op_r = 1 to wang.0
           var_1  =  substr(wang_string.lo_op_r,1,30)
           var_2  =  substr(wang_pos.lo_op_r,1,10)
           var_3  =  wang_name.lo_op_r
           say  var_1 var_2 var_3
        end
       end
     when re_sponse = 0 then
        say 'no matches found'
     Otherwise
        say 'Error :' re_sponse
  end
  reg_ex  = "(quick|jump)"
  opt_ion = "g"

  say reg_ex
  say the_str2

  re_sponse = REXXPCRE(reg_ex,the_str2,my_stem,opt_ion,,'debug')

  select
     when re_sponse = 1 then do
        say 'wang.0  is  ' wang.0
        do lo_op_r = 1 to wang.0
           var_1  =  substr(wang_string.lo_op_r,1,30)
           var_2  =  substr(wang_pos.lo_op_r,1,10)
           var_3  =  wang_name.lo_op_r
           say  var_1 var_2 var_3
        end
       end
     when re_sponse = 0 then
        say 'no matches found'
     Otherwise
        say 'Error :' re_sponse
  end


  say '  '

  say ' now the new version +++++++++++++++++++++++++++ '

  say '  '
*/
  my_stem = "wang"

  re_sponse = RXPCRE2('connect','pcre_env',,'debug')

  if re_sponse \= 0 then exit

  say 'pcre_env     = ' pcre_env

  re_sponse = RXPCRE2('compile',pcre_env,reg_ex,'pcre_comp',opt_ion)

  if re_sponse \= 0 then exit

  say 'pcre_comp    = ' pcre_comp

  re_sponse = RXPCRE2('compile',pcre_env,reg_ex_9,,
                      'pcre_comp_9',opt_ion_9)
```

```
if re_sponse \= 0 then exit

say 'pcre_comp_9  = ' pcre_comp_9

/*
   so now we have 2 seperate compilations which we should be
   able to use alternately
*/

say '  '
say reg_ex
say the_str1

/* note the double comma to tell rexx the parameters are
   continued on the next line                               */

pcre_exec = RXPCRE2('execute',pcre_env,pcre_comp,,
                    the_str1,my_stem)

say 'pcre_exec    = ' pcre_exec

say 'wang.0      is  ' wang.0
say 'wang_pos.0  is  ' wang_pos.0

select
   when pcre_exec = 1 then do
      say 'wang.0  is  ' wang.0
      do lo_op_r = 1 to wang.0
         var_1  =  substr(wang_string.lo_op_r,1,30)
         var_2  =  substr(wang_pos.lo_op_r,1,10)
         var_3  =  wang_name.lo_op_r
         say  var_1 var_2 var_3
      end
     end
   when pcre_exec = 0 then
      say 'no matches found'
   Otherwise
      say 'Error :' pcre_exec
end

say ' '
say reg_ex_9
say the_str9

pcre_exec_9 = RXPCRE2('execute',pcre_env,pcre_comp_9,,
                      the_str9,my_stem)

say 'pcre_exec_9  = ' pcre_exec_9
say 'wang.0      is   ' wang.0
say 'wang_pos.0  is   ' wang_pos.0

select
   when pcre_exec_9 = 1 then do
      say 'wang.0  is  ' wang.0
      do lo_op_r = 1 to wang.0
         var_1  =  substr(wang_string.lo_op_r,1,30)
         var_2  =  substr(wang_pos.lo_op_r,1,10)
         var_3  =  wang_name.lo_op_r
         say  var_1 var_2 var_3
      end
     end
   when pcre_exec_9 = 0 then
      say 'no matches found'
   Otherwise
      say 'Error :' pcre_exec_9
end

say ' '
say reg_ex
say the_str2


pcre_exec = RXPCRE2('execute',pcre_env,pcre_comp,the_str2,my_stem)

say 'pcre_exec    = ' pcre_exec

say 'wang.0      is   ' wang.0
say 'wang_pos.0  is   ' wang_pos.0

select
   when pcre_exec = 1 then do
      say 'wang.0  is  ' wang.0
      do lo_op_r = 1 to wang.0
         var_1  =  substr(wang_string.lo_op_r,1,30)
         var_2  =  substr(wang_pos.lo_op_r,1,10)
         var_3  =  wang_name.lo_op_r
         say  var_1 var_2 var_3
      end
     end
   when pcre_exec = 0 then
      say 'no matches found'
   Otherwise
      say 'Error :' pcre_exec
end

say ' '
say reg_ex_9
say the_str9a

pcre_exec_9 = RXPCRE2('execute',pcre_env,pcre_comp_9,,
                      the_str9a,my_stem)

say 'pcre_exec_9  = ' pcre_exec_9

say 'wang.0      is  ' wang.0
say 'wang_pos.0  is  ' wang_pos.0

select
   when pcre_exec_9 = 1 then do
      say 'wang.0  is  ' wang.0
      do lo_op_r = 1 to wang.0
         var_1  =  substr(wang_string.lo_op_r,1,30)
         var_2  =  substr(wang_pos.lo_op_r,1,10)
         var_3  =  wang_name.lo_op_r
         say  var_1 var_2 var_3
      end
     end
   when pcre_exec_9 = 0 then
      say 'no matches found'
   Otherwise
      say 'Error :' pcre_exec_9
end

say ' '
say reg_ex
say the_str3

pcre_exec = RXPCRE2('execute',pcre_env,pcre_comp,the_str3,my_stem)

say 'pcre_exec    = ' pcre_exec

say 'wang.0      is  ' wang.0
say 'wang_pos.0  is  ' wang_pos.0

select
   when pcre_exec = 1 then do
      say 'wang.0  is  ' wang.0
      do lo_op_r = 1 to wang.0
         var_1  =  substr(wang_string.lo_op_r,1,30)
         var_2  =  substr(wang_pos.lo_op_r,1,10)
         var_3  =  wang_name.lo_op_r
         say  var_1 var_2 var_3
      end
     end
   when pcre_exec = 0 then
```

```
         say 'no matches found'
      Otherwise
         say 'Error :' pcre_exec
   end

   say ' '
   say reg_ex
   say the_str4

   pcre_exec = RXPCRE2('execute',pcre_env,pcre_comp,the_str4,my_stem)

   say 'pcre_exec    = ' pcre_exec

   say 'wang.0      is   ' wang.0
   say 'wang_pos.0  is   ' wang_pos.0

   select
      when pcre_exec = 1 then do
         say 'wang.0  is  ' wang.0
         do lo_op_r = 1 to wang.0
            var_1  =  substr(wang_string.lo_op_r,1,30)
            var_2  =  substr(wang_pos.lo_op_r,1,10)
            var_3  =  wang_name.lo_op_r
            say  var_1 var_2 var_3
         end
        end
      when pcre_exec = 0 then
         say 'no matches found'
      Otherwise
         say 'Error :' pcre_exec
   end


   say ' '

   re_lease  = RXPCRE2('release',pcre_env,pcre_comp)

   say 're_lease     = ' re_lease
   say ' '

   re_lease_9 = RXPCRE2('release',pcre_env,pcre_comp_9)

   say 're_lease_9   = ' re_lease
   say ' '

   te_rm     = RXPCRE2('disconnect',pcre_env)

   say 'te_rm        = ' te_rm
   exit
```

RXPCREMB is a sample program that demonstrates usage of RXPCRE2 for substituting

```
/*REXX*/

  reg_ex     = "cat|dog"
  the_str    = "the dog's cat sat on the cat's dog"
  subs_str   = "horse"
  my_var     = "WANG"
  opt_ion    = "g"


  re_sponse = RXPCRE2('connect','pcre_env',,'debug')

  if re_sponse \= 0 then exit

  say 'pcre_env     = ' pcre_env

  re_sponse = RXPCRE2('compile',pcre_env,reg_ex,'pcre_comp1',opt_ion)

  if re_sponse \= 0 then exit

  say 'pcre_comp1   = ' pcre_comp1

  re_sponse = RXPCRE2('compile',pcre_env,reg_ex,'pcre_comp2')

  if re_sponse \= 0 then exit

  say 'pcre_comp2   = ' pcre_comp2


  say '  '
  say reg_ex
  say the_str
  say subs_str

  pcre_subs = RXPCRE2('subs',pcre_env,pcre_comp1,,
                      the_str,my_var,subs_str)

  say 'pcre_subs    = ' pcre_subs

  select
     when pcre_subs = 1 then do
         say 'output with -g is'
         say ':' || wang || ':'
       end
     when pcre_subs = 0 then
       say 'no matches found'
     Otherwise do
         say 'Error :' pcre_subs
         exit
       end
  end

  say ' '

  pcre_subs = RXPCRE2('subs',pcre_env,pcre_comp2,,
                      the_str,my_var,subs_str)

  say 'pcre_subs    = ' pcre_subs

  select
     when pcre_subs = 1 then do
         say 'output without -g is'
         say ':' || wang || ':'
       end
     when pcre_subs = 0 then
       say 'no matches found'
     Otherwise do
         say 'Error :' pcre_subs
         exit
       end
  end

  say ' '

  re_lease  = RXPCRE2('release',pcre_env,pcre_comp1)

  say 're_lease1    = ' re_lease

  re_lease  = RXPCRE2('release',pcre_env,pcre_comp2)

  say 're_lease2    = ' re_lease

  say ' '

  te_rm     = RXPCRE2('disconnect',pcre_env)

  say 'te_rm        = ' te_rm

  exit
```

Sample Stem Output:

The captured groups are presented in the stem. Let's consider this RegEx (not the best for the purpose)

```
Reg_Ex="(Mr\.\s(John|Alfred)\s(?:Sr\.|Jr\.))"
Paren # 1      2           2  3           31
```

We want to capture 'Mr. John Sr.', 'Mr. Alfred Jr.', etc. and we want to know whether it was John or Alfred. We do not care about John in 'John went to the game' as it is not prefixed with Mr. In the subject string **'Alfred went to visit Mr. John Jr.'** we want 'Mr. John Jr.' and 'John'. Let's count parentheses as demonstrated below the RegEx.
In stem_string.1 we'll find he whole match 'Mr. John Jr.', in 22,12.
In stem_string.2 we'll find the same since our all RegEx is the same as parentheses #1.
In Stem_string.2 we'll find 'John', in 26,4.
There won't be stem_string.4 because Parentheses #3 are non-capturing.
stem_...1 corresponds to the whole match, stem_...2 corresponds to captured group #1, and so on.
So the stem provides a vector of the matches and captured groups in the Rexx way!

Sample output:
We see above that we have to count parentheses in order to know where we are. This could become complicated.

```
(Mr\.\s(John|Alfred)\s(?:Sr\.|Jr\.))
Alfred went to visit Mr. John Jr. and Mr. Alfred Sr.
i   <----<<< option
3   <----<<< elements found

1   <----<<< the whole match group
Mr. John Jr.
22,12
2   <----<<< First Parentheses
Mr. John Jr.
22,12
3   <----<<< Second Parentheses
John
26,4
```

```
/\/\/\/\/\/\/\/\/\/\/\/
(Mr\.\s(John|Alfred)\s(?:Sr\.|Jr\.))
Alfred went to visit Mr. John Jr. and Mr. Alfred sr.
gi      <----<<< option, here we want to find all of them
6       <----<<< elements found, now it becomes complicated
1       <----<<< the whole match group; first time
Mr. John Jr.
22,12
2       <----<<< First Parentheses; first time
Mr. John Jr.
22,12
3       <----<<< Second Parentheses; first time
John
26,4
4       <----<<< the whole match group; second time
Mr. Alfred Sr.
39,14
5       <----<<< First Parentheses; second time
Mr. Alfred Sr.
39,14
6       <----<<< Second Parentheses; second time
Alfred
43,6
```

```
/\/\/\/\/\/\/\/\/\/\/\/
(?<paren1>Mr\.\s(?<paren2>John|Alfred)\s(?:Sr\.|Jr\.))
Alfred went to visit Mr. John Jr. and Mr. Alfred sr.
gi    <----<<< option
6     <----<<< elements found; the named groups help in
simplifying the result
1
Mr. John Jr.
22,12
2
Mr. John Jr.
22,12
paren1 <----<<< named group 1, 1st time
3
John
26,4
paren2 <----<<< named group 2, 1st time
4
Mr. Alfred Sr.
39,14
5
Mr. Alfred Sr.
39,14
paren1 <----<<< named group 1, 2nd time
6
Alfred
43,6
paren2 <----<<< named group 2, 2nd time
```

/\/\/\/\/\/\/\

Now the third parentheses are capturing but optional, without names this would be very hard to handle. Note, we found Alfred, but not the III

```
 (?<paren1>Mr\.\s(?<paren2>John|Alfred)\s(?<paren3>Sr\.|Jr\.)?)
Alfred went to visit Mr. John Jr. and Mr. Alfred III
gi       <----<<< option
7        <----<<< elements found

1
Mr. John Jr.
22,12

2
Mr. John Jr.
22,12
paren1

3
John
26,4
paren2

4
Jr.
31,3
paren3 <----<<< named group 3, 1st and only time

5
Mr. Alfred
39,11

6
Mr. Alfred
39,11
paren1

7
Alfred
43,6
paren2
```

References

- PCRE documentation: https://www.pcre.org/current/doc/html/
- RegExp, The Perl standard: https://perldoc.perl.org/perlre.html
- RegExp Tester: https://www.regexplanet.com/advanced/perl/index.html or https://regex101.com/ or https://www.regexpal.com/
- Posix BRE & ERE: https://en.wikibooks.org/wiki/Regular_Expressions/POSIX_Basic_Regular_Expressions and https://en.wikibooks.org/wiki/Regular_Expressions/POSIX-Extended_Regular_Expressions
- PCRE for z/OS (current version): http://cbttape.org/ftp/updates/CBT939.zip
- RegeExp Engines comparison: https://en.wikipedia.org/wiki/Comparison_of_regular-expression_engines
- The best printed book is "Mastering Regular Expressions" by Jeffery Friedl, 3rd Edition 2006, O'Reilly; it begins to age, but is still the best.
- Bug reports:
  - For misbehaving regular expressions use https://bugs.exim.org/enter_bug.cgi?product=PCRE. You will need to register (free). Please do not ask basic RegEx question and limit it to bug reports. Please be clear that you are referring to the z/OS port.
  - For bugs in the API (or the underlying port), you may drop me a line at zatlas1@yahoo.com. Please include the text "Bug Report for Rexx API" in the subject line. If I see a surge in such contacts I may consider a Bugzilla account as well.
  - If you are not sure, use the first contact to reach to a larger audience.

Your turn

Please consider helping this open source project. There are many ways that you could help such as:

- Suggest improvement in the API (add needed functionality, improved interface or the like.)
- Volunteer to actually code such improvements.
- Be aware that the API is coded in Assembler, using a version of Concept 14 macros, available on CBTTAPE, file 953. The actual port and supporting functions are coded in C. Coding functions in COBOL is theoretically possible as long as they are interfacing via the C modules. I have never went that route but would be open to try.

The project is configured as part of the CBTTAPE project (file 939) which is hosted on the Open Mainframe Project. I believe that the CBTTAPE would eventually get to GitHub or something similar in addition to the traditional EBCDIC/XMIT only distribution.