Rexx Regular Expression Library

Patrick TJ McPhee (ptjm@interlog.com)

Version 1.0.1, 26 May 2003

Contents

1 Introduction

This paper describes RexxRE, a set of functions which provide regular expression matching as defined in the POSIX specification. Regular expressions are commonly used in data parsing applications, and provide a useful extension to the Rexx function set, although some Rexx programmers consider the datatype( ) function to be an adequate replacement.

A regular expression (RE) is a string which represents some set of strings. The point of the regular expression routines is to search some target string for strings which can be represented by a regular expression. The simplest RE is a literal representation of exactly one string, for instance the RE ‘and’ represents the string ‘and’. More usefully, special characters can be used to broaden the set of strings represented by the RE. The RE ‘roo?t’ represents the strings ‘root’ and ‘rot’, while ‘[a-f0-9]{2,10}’ represents the set of hexadecimal numbers from 2 to 10 digits long. Section 2 gives a complete description of the regular expression syntax.

The POSIX regular expression interface defines four functions to perform regular expression matching. regcomp( ) converts a string representation of the regular expression into a format which can be efficiently compared to input strings. regexec( ) performs this comparison. regerror( ) returns error information in a human-readable format. regfree( ) releases the memory used to hold a compiled regular expression. Each of these functions has a direct analog in RexxRE. RexxRE includes an additional function which attempts to emulate the popular parse instruction using regular expressions.

On Unix systems, the regular expression routines are provided by the system. On Win32 and OS/2 platforms, I use the well-known regular expression routines by Henry Spencer of the University of Toronto, which are expected to have been brought into POSIX compliance by members of the CSRG at UC Berkley and by the FreeBSD project.

1.1 Installation

There is no installation program.

On Win32 platforms, copy win32/rexxre.dll to a directory on your path, or the directory containing regina.exe or your rexx-enabled application. If you use a Rexx interpreter other than Regina, you either need to recompile the library using your interpreter’s development kit or download the rexx/trans version of RexxRE.

On OS/2, copy os2/rexxre.dll to a directory in your LIBPATH.

On Unix, you need to compile the library. The distribution does not include a configuration script, but it includes make files which have been known to work using the stock vendor compiler on several Unix systems. If you have one of those systems, link the appropriate make file to the name ‘Makefile’ and build the ‘dist’ target. For instance, on Solaris:

```
ln Makefile.sun Makefile
make dist
```

On most platforms, this builds a shared library called librexxre.so. On HP-UX, the file is called librexxre.sl, and on AIX, it's called librexxre.a. The path to this library can be set in three ways:

Most Unix systems allow a shared library search path to be embedded into program files. If you build regina (or your rexx-enabled application) such that this path is set to include a directory such as /opt/regina/lib or /usr/local/lib, you can install RexxRE by copying the shared library to this directory. Otherwise, you need to either set an environment variable or change the way the system searches for shared libraries.

Unix systems typically use a different path for shared libraries than they do for program files. The name of the environment variable used for the shared library path is not standardised, however most systems use LD_LIBRARY_PATH. Notable exceptions are AIX (LIBPATH) and HP-UX (SHLIB_PATH for 32-bit executables, LD_LIBRARY_PATH for 64-bit executables). To install RexxRE, add an appropriate directory to the shared library path for your machine and copy the shared library to that directory.

Finally, some systems provide a utility (often called ldconfig) which can be used either to set the standard search path for shared libraries, or to provide a database of shared libraries. On such a system, RexxRE can be installed by copying the shared library to an appropriate directory and using this utility to add it to the search database. You'll need to consult your system documentation for more information.

1.2 Reporting Bugs

I don't anticipate making a lot of changes to this library in the future, but I would like it to be bug-free.

If you find a bug, an error in the documentation, or you simply have a suggestion for improving the distribution, please send me details at ptjm@interlog.com. It's useful to know the operating system you're using, the version of Regina (or Rexx/IMC), and the version of rexxre, and to have a set of steps for reproducing the bug.

1.3 Using RxFuncAdd

All the routines in rexxre can be loaded either directly using RxFuncAdd, or indirectly using ReLoadFuncs. RxFuncAdd takes three arguments – the name of the function as it will be used in the rexx program, the name of the library from which to load the function, and the name of the function as it appears in the library.

RxFuncAdd returns 0 on success, or 1 on failure. Regina has a function called RxFuncErrMsg which can give useful information about the reason for a load failure. A few common reasons for failure are:

Path issues: the library is called rexxre.dll on Win32 and OS/2 platforms, librexxre.a on AIX, librexxre.sl on HP-UX, and librexxre.so on other Unix platforms. On Win32, this file needs to be in the path, or in the directory containing regina.exe. On OS/2, it needs to be in a directory specified in LIBPATH. On Unix systems, it needs to be in a directory listed in LIBPATH on AIX, SHLIB_PATH on HP-UX 32-bit, or LD_LIBRARY_PATH on most other Unix systems. Some systems have an ldconfig utility which allows you to forego setting this environment variable.

Windows 95: early releases of windows 95 did not include msvcrt.dll, the C runtime library used by RexxRE. This library is sometimes installed with applications software. It can also be obtained through service packs, or from the Microsoft web site.

Rexx.exe: Regina includes two executables, one called ‘rexx’, and the other called ‘regina’. The difference is that ‘rexx’ includes the Rexx interpreter as part of the executable, while ‘regina’ loads the interpreter from a shared library. RxFuncAdd works only with the ‘regina’ version of the interpreter (the ‘rexx’ version is slightly faster, though).

1.4 Licencing

RexxRE is distributed free of charge in the hopes that it will be useful, but without any warranty. It is distributed under the terms of the Mozilla Public License. The precise details of the licence are found in the file MPL-1.1.txt in the distribution. Henry Spencer’s regular expression routines are taken from the FreeBSD distribution and are distributed under the BSD licence.

2 Regular Expressions

This section describes the regular expression (RE) format defined by the POSIX standard. Because the regular expression processing is actually handled by operating-system-specific functions, this syntax might not exactly match the syntax available on your system, and there’s little I can do about it. Most current systems should work as described here. For others, you might try compiling with the Spencer routines found in the regex sub-directory.

For historical reasons, there are two regular expression syntaxes defined by POSIX, which are usually called ‘basic’ and ‘extended’. We are encouraged to use ‘extended’ REs, however most Unix utilities actually use ‘basic’ REs, and the default format for the regular expression routines is ‘basic’. To make things more confusing, many Unix utilities extend either type of RE in one way or another, many editors provide their own unique regular expression syntax, the regular expression library included with Object Rexx is similar to, but different from, ‘extended’ regular expressions, and some people are familiar with the regular expressions provided by perl, which are syntactically completely different from anything else anywhere. I have elected to stick with the POSIX definitions since they are the only reasonable standard.

2.1 Common Syntax

Both ‘basic’ and ‘extended’ regular expressions consist of strings of characters, some of which have special meanings. The regular expression is expressed in terms of ‘ordinary’ characters, which correspond to themselves, and ‘meta’ characters, which generally change the way the regular expression matching is performed.

The set of ordinary and meta characters differs between ‘basic’ and ‘extended’ regular expressions, however there is some common ground. In general, non-punctuation characters are always ordinary. ‘cat’ matches only the string ‘cat’, and ‘12’ matches only ‘12’.

The characters below are meta characters in both types of RE. Later sections list the meta characters which are specific to each type of RE.

. Matches any character. ‘c.t’ matches ‘cat’, ‘cot’, ‘cut’, ‘ctt’, *etc.*;

[Introduces an alternation set. ‘c[auo]t’ matches ‘cat’, ‘cut’, or ‘cot’, but not ‘ctt’ or any of the other things matched by ‘c.t’. ‘^’, ‘-’, and ‘]’ can have special meanings in alternation sets. Alternation is discussed more fully in section 2.4;

* Kleene closure: matches 0 or more of whatever the preceding character or group of characters matches. ‘.*’ matches any string. ‘a*’ matches ‘’, ‘a’, ‘aa’, *etc.*. The regular expression engine will take the first, longest match for Kleene closure. ‘a*’ will match the first three characters of ‘aaabcaa’;

\ Escapes itself and the other meta characters. \. matches ‘.’ only. In ‘basic’ regular expressions, \ also causes some ordinary characters to act like meta characters;

^ When placed at the beginning of a regular expression, anchors the regular expression to the start of the string being searched. The regular expression ‘at’ matches the last two characters of ‘cat’, but ‘^at’ does not. Elsewhere in the regular expression, ^ has no special meaning;

$ When placed at the end of a regular expression, anchors the regular expression to the end of the string being searched. The regular expression ‘ca’ matches the first two characters of ‘cat’, but ‘ca$’ does not.

2.2 Basic Regular Expressions

‘Basic’ regular expressions use \ to turn some ordinary characters into meta characters.

\(Begins a group. The regular expression which appears between \(and \) is treated as a unit with respect to Kleene closure and interval operations. ‘\(abc\)*’ matches ‘’, ‘abc’, ‘abcabc’, *etc.*; Group contents can also be returned to the caller by the regular expression matching functions, and can be used in back-references;

\) Ends a group;

\d Back-reference: matches the contents of the *d*th group, where *d* is an integer from 1 to 9. ‘\([cd]\)at\1’ matches ‘catcat’ and ‘datdat’, but not ‘datcat’;

\{ Interval: \{must be followed by one of a whole number w_1, a whole number w_2 and a comma, or two whole numbers w_3 and w_4, $w_3 < w_4$ separated by a comma. The interval constrains the preceding character or group to match exactly w_1 instances, w_2 or more instances, or between w_3 and w_4 instances, inclusive, depending on the syntax used. ‘a\{3\}’ matches the first three characters of ‘aaaaaa’. ‘a\{3,\}’ matches all the characters of ‘aaaaaa’ but does not match ‘aa’. ‘a\{3,4\}’ matches the first four characters of ‘aaaaaa’ but does not match ‘aa’.

\} Closes an interval.

2.3 Extended Regular Expressions

‘Extended’ regular expressions introduce a few new meta characters which make certain kinds of regular expression more convenient to type. They eliminate back-references, which makes other kinds of regular expression impossible.

+ Kleene closure. matches 1 or more of whatever the preceding character or group of characters matches. ‘.+’ matches any string of non-zero length. ‘a+’ matches ‘a’, ‘aa’, *etc*.. The regular expression engine will take the first, longest match for Kleene closure. ‘a+’ will match the first three characters of ‘aaabcaa’;

? Matches 0 or 1 of whatever the preceding character or group of characters matches. ‘^noodles?$’ matches ‘noodle’ or ‘noodles’;

(Begins a group. The regular expression which appears between (and) is treated as a unit with respect to Kleene closure and interval operations. ‘(abc)*’ matches ‘’, ‘abc’, ‘abcabc’, *etc*.; Group contents can also be returned to the caller by the regular expression matching functions;

) Ends a group;

| Branching: divides the regular expression (or group within a regular expression) into alternatives. The regular expression will match the first, longest match of any of the alternatives. ‘cat|dog’ will match either ‘cat’ or ‘dog’. It will match the first three characters of ‘cats and dogs’. ‘^ *(int|long) +[a-z_][a-z0-9A-Z_]*;’ matches the declaration of an int or long variable in a C program;

{ Interval: {must be followed by one of a whole number w_1, a whole number w_2 and a comma, or two whole numbers w_3 and w_4, $w_3 < w_4$ separated by a comma. The interval constrains the preceding character or group to match exactly w_1 instances, w_2 or more instances, or between w_3 and w_4 instances, inclusive, depending on the syntax used. ‘a{3}’ matches the first three characters of ‘aaaaaa’. ‘a{3,}’ matches all the characters of ‘aaaaaa’ but does not match ‘aa’. ‘a{3,4}’ matches the first four characters of ‘aaaaaa’ but does not match ‘aa’. Note that Object Rexx supports only the first interval syntax;

} Closes an interval.

2.4 Alternation Sets

Alternation is a useful but seemingly confusing aspect of regular expression syntax. At its simplest, a set of characters delimited by [and] matches any character in the set. [abc] matches any of a, b, or c. The alternation set is treated as a single unit for the purposes of Kleene closure and interval operations.

Alternation sets have a different set of meta characters from the rest of the RE. For instance, '.' is an ordinary character within an alternation set, but '^', '-', and ']' have special meanings.

^ When it appears at the very start of the alternation set, '^' inverts the set. '[^abc]' matches any character *but* a, b, or c. When it appears elsewhere in the set, it has no special meaning;

- When it appears between two other characters, - represents all the characters which appear between those two characters in the current character set. On ASCII systems, '[a-z]' represents any lower-case letter. - represents itself when placed at either the start or the end of the alternation set. '[^-]' represents any character but -. '[-+/*]' represents any arithmetic operator;

] When it appears anywhere but at the start of the alternation set,] terminates the set. '[^]]' represents any character but]. '[]a-z]' represents any lower-case letter or].

Since the use of character values and the use of the range operator results in definitions which are character-set- and natural-language-specific, POSIX defines classes of characters which can be used by name within alternation sets, by enclosing the names within '[:' and ':]'. These classes are:

alnum	letters and numbers;
alpha	letters;
blank	space or tab;
cntrl	control characters;
digit	numbers;
graph	printable characters;
lower	lower-case letters;
print	printable characters;
punct	punctuation;
space	white space (including newlines, vertical tabs, *etc.*);
upper	upper-case letters;
xdigit	hexadecimal digits.

A C language identifier consists of a letter or underscore followed by any number of letters, digits, or underscores. This could be matched by the regular expression '[a-zA-Z_][a-zA-Z0-9_]*', however this will work only on machines that use the ASCII character set. The same regular expression using character classes is '[[:alpha:]_][[:alnum:]_]*'.

Note that the POSIX treatment of character classes is different from the Object Rexx treatment.

3 Function Definitions

3.1 Introduction

Normal use of RexxRE is to register the library using ReLoadFuncs( ), compile one or more regular expressions using ReComp( ), then match the compiled regular expressions against input strings using ReExec( ). Alternatively, ReParse( ) can be used to perform an operation similar to the parse value instruction, using regular expressions for the parse template.

Compiling regular expressions is optional, but you should note that there is no way to determine whether the regular expression was compiled correctly by either ReExec( ) or ReParse( ), and compiling regular expressions provides a performance gain unless the regular expression match is performed only once.

If an error does occur while compiling the regular expression, useful diagnostic may be retrieved by calling ReError( ). Once the compiled regular expression is no longer needed, it can be released using ReFree( ).

3.2 ReLoadFuncs

```
ReLoadFuncs()
```

ReLoadFuncs( ) registers all the other routines in RexxRE with the Rexx interpreter. This registration takes less work on your part than registering each individual function using RxFuncAdd( ), but you can register the functions you intend to use individually if that makes you feel better.

3.3 ReDropFuncs

```
ReDropFuncs()
```

ReDropFuncs( ) unregisters all the routines in RexxRE. It is safe to do this even if the functions were not registered using ReLoadFuncs( ). With Regina and likely most other interpreters, function registration ends when the Rexx process terminates in any case, however with IBM's interpreters, function registration persists until the functions are unregistered or the machine is rebooted, so it's good practice to call ReDropFuncs( ) at the end of each program.

3.4 ReVersion

```
ReVersion() -> version string
```

ReVersion( ) reports the version number of the library in the format *version*.*release*.*modification*. It's often useful to have this information when investigating bugs.

Typically, I change the *modification* level whenever I make a bug-fix release, I change the *release* number whenver I add new functions, and I change the *version* number when I add dramatically new functionality. This might happen if I elected to support the Object Rexx regular expression syntax, for instance.

3.5 ReComp

```
ReComp(re, flags) -> cre
```

ReComp( ) compiles the regular expression *re* into the compiled regular expression *cre*. By default, *re* is a ‘basic’ regular expression, and it is compiled to match case-insensitively, meaning lower-case letters in *re* will match the corresponding upper- and lower-case letters.

You need to call ReComp( ) only once per regular expression, and you can have any number of compiled regular expressions active at any time. Both ReExec( ) and ReParse( ) accept uncompiled regular expressions as well as compiled regular expressions, so you may leave out the compilation step if you don’t need to perform repeated matches against the same RE.

You should release the memory allocated for the regular expression by calling ReFree( ) when you no longer need it.

Flags can be any combination of

- x means *re* is an ‘extended’ regular expression;
- c to get case-sensitive matches
- s means that only the status of the match will be returned from ReExec( ). When this flag is given, ReExec( ) will return as soon as some match of the regular expression has been achieved. Normally the regular expression engine will continue processing until it finds the first, longest match, and it returns either the matched strings or their position. Do not pass the *matches* argument to ReExec( ) if you pass the ‘s’ flag to ReComp( ), and do not use a compiled regular expression which used the ‘s’ flag as an argument to ReParse( );
- n Treat new-lines as match delimiters. Normally, a new-line in a target string is treated like any other control character, but when the regular expression is compiled with the ‘n’ flag, new-lines are never matched by the regular expression and are used to delimit RE anchors (*e.g.,* ‘cat$’ would match ‘cat[newline]’).

If the regular expression compiles successfully, the first character of *cre* will be 0. Otherwise, it will be 1. You can test it like this:

```
cre = ReComp('cat$', 'c')
if left(cre, 1) then do
  say 'failed to compile RE cat$:' ReError(cre)
  call ReFree cre
  exit 1
  end
```

3.6 ReExec

```
ReExec(cre, string[, matches[, flags]]) -> 0 or 1
```

ReExec( ) searches the target string *string* for the first, longest substring which can be represented by the regular expression *cre*. *cre* ought usually to have been compiled by a previous call to ReComp, however it can also be a string representation of the RE, at the cost of performance.

matches is the name of a stem used to store information about the match. On successful return, *matches*.0 contains the number of parenthesized sub-expressions of *cre*. Each of *matches*.1 to *matches*.(*matches*.0) contains the string that matches the corresponding parenthesised subexpression of *cre*. *matches*.!match contains the string which matches the whole of *cre*. Do not pass a value for *matches* if *cre* was compiled with the 's' flag.

flags can be any combination of the flag arguments to ReComp( ) and these values:

p instead of returning strings in *matches*, return the offset and length of each match within *string*, separated by a single space;

b the string does not start at the beginning of a line. This affects ^ processing;

e the string does not end at the end of a line. This affects $ processing.

ReComp( ) flags are used only when compiling a string-format regular expression. I don't advise doing this as it's usually faster to use the result of ReComp( ), and there's no way to test for errors in regular expression compilation when it's done by ReExec( ). In any case, passing, for instance 'c' as a flag to ReExec( ) will have no effect if *cre* is a compiled regular expression.

ReExec returns 1 if a match was found for the regular expression, and 0 otherwise.

3.7 ReError

```
ReError(cre) -> error string
```

ReError( ) takes a compiled regular expression returned from a failed call to ReComp( ) and returns a string which explains the problem.

3.8 ReFree

```
ReFree(cre[, cre, ...])
```

ReFree( ) releases memory used to hold the compiled regular expressions *cre*. For complicated regular expressions, memory usage can be substantial, and it makes sense to call ReFree( ) as soon as you no longer need the RE.

3.9 ReParse

```
ReParse(cre, string, [flags], varname[, varname, ...]) -> 0 or 1
```

ReParse( ) attempts to provide ‘parse value’-like functionality using regular expressions. The arguments are a regular expression, which should normally have been compiled by ReComp( ), a target string, flags that affect the processing of the regular expression, and a list of variable names. The regular expression is used to divide the target string into fields, which are then assigned to the variables as described below.

By default, *cre* represents a field delimiter, and each field is assigned to the corresponding variable. As with the parse instruction, if more data is available than there are variable names, all the data at the end of *string* is written to the final variable in the list. These lines should be equivalent

```
call ReParse '  *', value, , a, b, c
parse value value with a b c
```

Flags can be any combination of the flag arguments to ReComp( ) and these values, which determine how fields are assigned to variables:

d *cre* represents the delimiter between each field. If the delimiter appears at the start of *string*, then the first field has zero length. If the number of variables is less than then number of delimited fields, the last variable in the list is assigned the remaining portion of the string, excluding the leading delimiter. If the number of variables is greater than the the number of delimited fields, all the unmatched variables are set to zero-length. This is the default;

v *cre* represents the format of a field. If each field consists of integer data, *cre* might be ‘[0-9]+’. If *n* variable names are passed as arguments, up to the first $n-1$ field values are assigned to the corresponding variables from the list. The *n*th variable is always assigned the portion of *string* which remains after the field assignments. It can then be used for further parse operations. If there are fewer than *n* fields, any variables which do not correspond to a field are set to zero length;

s each field corresponds to a parenthesised sub-expressions. If the number of variables is less than the number of sub-expressions, the extra matches will be thrown away. Note that this is essentially the opposite meaning to the ‘s’ flag of ReComp( );

t *varname* is the name of a stem variable, and fields are written to the stem using the numeric convention.

The ReComp( ) flags are used only if *cre* is a string representation of the RE. You should compile any regular expression which will be used in the parse or exec routines more than once, since the performance is better. You should also compile any regular expression which is created dynamically or input by the end user, in order to validate its compilation.

3.9.1 Examples

Parsing a syslog entry:

```
/* here’s a syslog entry */
data = ’Apr 29 11:55:17  su: ptjm to root on /dev/ttyp2’

/* this RE matches some characters which are not :, a space,
 * two digits forming sub-expression 1, a colon, two digits
 * forming sub-expression 2, a colon, two digits forming
 * sub-exression 3, any number of spaces, some bunch of
 * non-colon characters forming sub-expression 4, a colon, a space,
 * and whatever’s left in the target string, forming sub-expression 5 */
re = ’[^:]+ ([0-9]{2}):([0-9]{2}):([0-9]{2}) +([^:]+): (.+)’

/* the parse call matches the sub-expressions to the variables
 * hh, mi, ss, component, and msg, respectively */
call reParse re, data, ’xs’, ’hh’, ’mi’, ’ss’, ’component’, ’msg’
/* now, hh = 11; mi = 55; ss = 17; component = ’su’
        msg = ’ptjm to root on /dev/ttyp2’ */

/* this RE matches any number of spaces and colons */
re = ’[ :]{1,}’

/* the parse call matches each space-delimited expression to a
 * variable */
call reParse re, data, , ., ., ’hh’, ’mi’, ’ss’, ’component’, ’msg’
/* now, hh = 11; mi = 55; ss = 17; component = ’su’
        msg = ’ptjm to root on /dev/ttyp2’ */

/* this RE matches any number of alpha-numerics */
rean = ReComp(’[[:alnum:]]+’, ’x’)

/* this one matches any number of numerics */
ren = ReComp(’[[:digit:]]+’, ’x’)

/* this one matches any number of non-spaces or colons */
re!sc = ReComp(’[^ :]+’, ’x’)

/* throw away the date */
call reParse(rean, data, ’v’, ’.’,  ’.’, data)
/* data = ’ 11:55:17  su: ptjm to root on /dev/ttyp2’ */

/* get the time */
call reParse(ren, data, ’v’, ’hh’, ’mi’, ’ss’, data)
/* data = ’  su: ptjm to root on /dev/ttyp2’ */

/* get the component */
call reParse(re!sc, data, ’v’, ’component’, data)
```

```
  /* data = ': ptjm to root on /dev/ttyp2' */
  msg = substr(data, 2)
  /* now, hh = 11; mi = 55; ss = 17; component = 'su'
          msg = 'ptjm to root on /dev/ttyp2' */
  call ReFree ren, rean, re!sc
```

Flexible date processing:

```
parsedate: procedure
mwre = ReComp('[[:alpha:]]+', 'x')
mdre = ReComp('[[:digit:]]{1,2}', 'x')
yre = ReComp('[[:digit:]]{4}', 'x')

parse arg date

pdate = date
rdate = 'baddate'

/* test for April 30 2003 */
if ReParse(mwre, pdate, 'v', 'month', 'pdate') then do
    if ReParse(mdre, pdate, 'v', 'day', 'pdate') then do
        if ReParse(yre, pdate, 'v', 'year', 'pdate') then
             rdate = year'/'mwords(month)'/'day
        end
    /* start over, trying 30 April 2003 */
    if rdate = 'baddate' then do
        pdate = date
        if ReParse(mdre, pdate, 'v', 'day', 'pdate') then do
            if ReParse(mwre, pdate, 'v', 'month', 'pdate') then do
                if ReParse(yre, pdate, 'v', 'year', 'pdate') then
                     rdate = year'/'mwords(month)'/'right(day,2,'0')
                end
            end
        end
    end
else do
    /* test for 2003/04/30 */
    pdate = date
    if ReParse(yre, pdate, 'v', 'year', 'pdate') then do
        if ReParse(mdre, pdate, 'v', 'month', 'pdate') then do
            if ReParse(mdre, pdate, 'v', 'day', 'pdate') then
                rdate = year'/'right(month,2,'0')'/'right(day,2,'0')
            end
        end

    /* test for 30/04/2003, then give up */
    if rdate = 'baddate' then do
        pdate = date
        if ReParse(mdre, pdate, 'v', 'day', 'pdate') then do
            if ReParse(mdre, pdate, 'v', 'month', 'pdate') then do
```

```
                if ReParse(yre, pdate, ’v’, ’year’, ’pdate’) then
                    rdate = year’/’right(month,2,’0’)’/’right(day,2,’0’)
                end
            end
        end
    end

call ReFree mwre, yre, mdre
return rdate
```