Comparing namedCapture with other R packages for regular expressions

by Toby Dylan Hocking

Abstract Regular expressions are powerful tools for manipulating non-tabular textual data. For many tasks (visualization, machine learning, etc), tables of numbers must be extracted from such data before processing by other R functions. We present the R package **namedCapture**, which facilitates such tasks by providing a new user-friendly syntax for defining regular expressions in R code. We begin by describing the history of regular expressions and their usage in R. We then describe the new features of the namedCapture package, and provide detailed comparisons with related R packages (**rex**, **stringr**, **stringi**, **tidyr**, **rematch2**, **re2r**).

Introduction

Today regular expression libraries are powerful and widespread tools for text processing. A regular expression *pattern* is typically a character string that defines a set of possible *matches* in some other *subject* strings. For example the pattern `o+` matches one or more lower-case o characters; it would match the last two characters in the subject `foo`, and it would not match in the subject `bar`.

The focus of this article is regular expressions with capture groups, which are used to extract subject substrings. Capture groups are typically defined using parentheses. For example, the pattern `[0-9]+` matches one or more digits (e.g. `123` but not `abc`), and the pattern `[0-9]+-[0-9]+` matches a range of integers (e.g. `9-5`). The pattern `([0-9]+)-([0-9]+)` will perform matching identically, but provides access by number/index to the strings matched by the capturing sub-patterns enclosed in parentheses (group 1 matches `9`, group 2 matches `5`). The pattern `(?P<start>[0-9]+)-(?P<end>[0-9]+)` further provides access by name to the captured sub-strings (`start` group matches `9`, `end` group matches `5`). In R named capture groups are useful in order to create more readable regular expressions (names document the purpose of each sub-pattern), and to create more readable R code (it is easier to understand the intent of named references than numbered references).

We begin by providing a brief history of regular expressions and their usage in R. We then provide an overview of current R packages for regular expressions.

Origin of regular expressions and named capture groups

Regular expressions were first proposed on paper by Kleene (1956). Among the first uses of a regular expression in computers was for searching in a text editor (Thompson, 1968) and lexical processing of source code (Johnson et al., 1968).

A capture group in a regular expression is used to extract text that matches a sub-pattern. In 1974, Thompson wrote the `grep` command line program, which was among the first to support capture groups (Friedl, 2002). In that program, backslash-escaped parentheses `\(\)` were used to open and close each capture group, which could then be referenced by number (`\1` for the first capture group, etc).

The idea for named capture groups seems to have originated in 1994 with the contributions of Tracy Tims to Python 1.0.0, which used the `\(<labelname>...\)` syntax (Python developers, 1997a). Python 1.5 introduced the `(?P<labelname>...)` syntax for name capture groups (Python developers, 1997b); the P was used to indicate that the syntax was a Python extension to the standard.

Perl-Compatible Regular Expressions (PCRE) is a C library that is now widely used in free/open-source software tools such as Python and R. PCRE introduced support for named capture in 2003, based on the Python syntax (Hazel, 2003). Starting in 2006, it supported the `(?P<labelname>...)` and `(?'labelname'...)` syntax, to be consistent with Perl and .NET (Hazel, 2003).

The first regular expression support in R was provided by the TRE C library (Laurikari, 2019). Although TRE supports capture groups, it does not allow capture groups to be named. PCRE was first included in R version 1.6.0 in 2002 (R Core Team, 2002). The base R functions `regexpr` and `gregexpr` can be given the `perl=TRUE` argument in order to use the PCRE library, or `perl=FALSE` to use the TRE C library. Recently created packages (**stringi**, **re2r**) have provided R interfaces to the ICU and RE2 libraries. Each library has different characteristics in terms of supported regex features and time complexity (Table 1).

The original versions of `regexpr` and `gregexpr` only returned the position/length of the text matched by an entire regex, not the capture groups (even though this is supported in TRE/PCRE). I

C library	RE2	PCRE	ICU	TRE
Named capture groups	yes	yes	no	no
Worst case linear time	yes	no	no	no
Backreferences	no	yes	yes	yes

Table 1: Features of C libraries for regular expressions usable in R.

wrote the C code that uses PCRE to extract the text matched by each named capture group (Hocking, 2011a), which was accepted into R starting with version 2.14. I presented a lightning talk at useR 2011 that showcased the new functionality (Hocking, 2011b).

Related R packages for capturing regular expressions

Since the introduction of named capture support in base R version 2.14, several packages have been developed which use this functionality, and other packages have been developed which use other C libraries (Table 2). Each package supports different options for subject/pattern input, extracted text outputs, named capture groups, and type conversion (Table 3).

The **utils** package now includes the `strcapture` function, which uses the base `regexec` function (also introduced in R-2.14) to extract the first match as a data.frame with one row per subject, and one column per capture group. It allows capture group names/types to be specified in a prototype data.frame argument, but does not allow capture group names in the regex pattern. PCRE is used with `perl=TRUE` and TRE is used with `perl=FALSE`.

The **rematch2** package provides the `re_match` function which extracts the first match using the base `regexpr` function (Csárdi, 2017). It also provides the `re_match_all` function which extracts all matches using the base `gregexpr` function. In both cases the output is a tibble (a data.frame subclass) with one row for each subject (for all matches a list column is used). PCRE is used with `perl=TRUE` and TRE is used with `perl=FALSE`. Although TRE supports capture groups (and can be used via the base R `regexec` function), capture groups are not supported in rematch2 with `perl=FALSE` (because it uses the base `regexpr`/`gregexpr` functions which do not return group info for TRE). Named capture groups are supported in rematch2 with `perl=TRUE`.

The **stringi** package provides the `stri_match` and `stri_match_all` functions, which use the ICU C library (Gagolewski, 2018). The **stringr** package provides the `str_match` and `str_match_all` functions, which simply call the analogous functions from **stringi**. Capture groups are supported but named groups are not, so groups must be extracted by number. The `stri_match` function returns a character matrix with one row for each subject and one column for each capture group. The `stri_match_all` function returns a list with one element for each subject; each element is a data frame with one row for each match, and one column for each capture group.

The **re2r** package provides the `re2_match` and `re2_match_all` functions, which use the RE2 C++ library (Qin Wenfeng, 2017). The outputs of these functions are consistent with the **stringi**/**stringr** packages. The input regex pattern may be specified as a character string or as a pre-compiled regex object (which results in faster matching if the regex is used with several calls to matching functions). Like TRE, the RE2 library guarantees linear time complexity, which is useful to avoid denial-of-service attacks from malicious patterns (see Section "Comparing computation times of R regex packages").

The **rex** package provides the `re_matches` function which supports named capture groups, and always uses PCRE (Ushey et al., 2017). By default it returns the first match (using the base `regexpr` function), as a data.frame with one row for each subject, and one column for each capture group. If the `global=TRUE` argument is given, `gregexpr` is used to return all matches as a list of data.frames. A unique feature of the **rex** package is a set of functions for defining a regular expression in R code, which is then converted to a standard PCRE regex pattern string (for a detailed comparison with the proposed syntax of the **namedCapture** package, see Section "Comparing **namedCapture** variable argument syntax with **rex**").

The **tidyr** package provides the `extract` function which uses the ICU library, so does not support regex patterns with named capture groups (Wickham and Henry, 2018). The subject is specified via the first two arguments: (1) a data.frame, and (2) a column name. The pattern is specified via the second two arguments: (3) a character vector for the capture group names, and (4) the regex pattern string (it is an error if the number of capture group names does not match the number of un-named capture groups in the regex pattern). The pattern is used to find the first match in each subject. The return value is a data.frame with the same number of rows as the input, but without the subject column, and with an additional column for each capture group.

Package	First match	All matches	C library
base	`regexpr`	`gregexpr`	PCRE/TRE
utils	`strcapture`	NA	PCRE/TRE
rematch2	`re_match`	`re_match_all`	PCRE/TRE
namedCapture	`str_match_*`, `df_match_variable`	`str_match_all_*`	PCRE/RE2
rex	`re_matches(global=FALSE)`	`re_matches(global=TRUE)`	PCRE
stringr	`str_match`	`str_match_all`	ICU
stringi	`stri_match`	`stri_match_all`	ICU
tidyr	`extract`	NA	ICU
re2r	`re2_match`	`re2_match_all`	RE2

Table 2: R packages that provide functions for extracting first/all regex matches, and C library used.

Package/function	subject	pattern	outputs	named	types
base	chr	chr	mat/list	yes	no
`utils::strcapture`	chr	chr	df	no	some
rematch2	chr	chr	tibble	yes	no
namedCapture	chr/df/dt	verbose	mat/list/df/dt	yes	any
rex	chr	verbose	df/list	yes	no
stringr	chr	chr	mat/list	no	no
stringi	chr	chr	mat/list	no	no
`tidyr::extract`	df/dt	chr	df/dt	no	some
re2r	chr	chr/compiled	df/list	yes	no

Table 3: R packages provide different options for subject/pattern input, extracted text outputs, named capture groups, and type conversion.

The namedCapture package

The **namedCapture** package provides functions for extracting numeric data tables from non-tabular text data using named capture regular expressions. By default, **namedCapture** uses the RE2 C library if the **re2r** package is available, and PCRE otherwise (via the base `regexpr` and `gregexpr` functions). RE2 is preferred because it is guaranteed to find a match in linear time (see Section Comparing computation times of R regex packages). However, PCRE supports some regex features (e.g. backreferences) that RE2 does not. To tell **namedCapture** to use PCRE rather than RE2, `options(namedCapture.engine="PCRE"` can be specified. For patterns that are supported by both engines, **namedCapture** functions return the resulting match in the standard output format described below.

The main design features of the namedCapture package are inspired by the base R system, which provides good support for naming objects, and referring to objects by name. In particular, the namedCapture package supports

- Specifying capture groups with names in a regular expression string, and stopping with an informative error if there are un-named capture groups.
- Output with rownames or list names taken from the `name` capture group.
- A syntax for specifying capture group names via named arguments in R code.
- Specifying a function for each named capture group, which converts captured text from character to other arbitrary types.
- Saving sub-patterns to R variables, and re-using them multiple times in one or several patterns in order to avoid repetition.

The main functions provided by the namedCapture package are summarized in Table 4. We begin by introducing the `*_named` functions, which take three arguments.

Three argument syntax: `str_match_named` and `str_match_all_named`

The most basic functions of the **namedCapture** package are `str_match_named` and `str_match_all_named`, which accept exactly three arguments:

- subject: a character vector from which we want to extract tabular data.

First match	All matches	Arguments
`str_match_named`	`str_match_all_named`	chr subject, chr pattern, functions
`str_match_variable`	`str_match_all_variable`	chr subject, chr/list/function, ...
`df_match_variable`	NA	df subject, chr/list/function, ...

Table 4: Functions of the namedCapture package. The first argument of each function specifies the subject, as either a character vector (for `str_*`) functions, or a data.frame (for `df_match_variable`). The `*_named` functions require three arguments, whereas the `*_variable` functions take a variable number of arguments.

- pattern: the (character scalar) regular expression with named capture groups used for extraction.
- fun.list: a list with names that correspond to capture groups, and values are functions used to convert the extracted character data to other (typically numeric) types.

For an example, we consider subjects containing genomic positions:

```
> chr.pos.subject <- c(
+   "chr10:213,054,000-213,055,000",
+   "chrM:111,000",
+   "this will not match",
+   NA, # neither will this.
+   "chr1:110-111 chr2:220-222") # two possible matches.
>
```

These subjects consist of a chromosome name string, a colon, a start position, and optionally a dash and and end position. The following pattern is used to extract those data:

```
> chr.pos.pattern <- paste0(
+   "(?P<chrom>chr.*?)",
+   ":",
+   "(?P<chromStart>[0-9,]+)",
+   "(?:",
+     "-",
+     "(?P<chromEnd>[0-9,]+)",
+   ")?")
>
```

The pattern above is defined using `paste0`, writing each named capture group on a separate line, which increases readability of the pattern. By default the `str_match_named` function returns a character matrix with one row for each subject and one column for each capture group. Column names are taken from the group names that were specified in the regular expression pattern:

```
> (match.mat <- namedCapture::str_match_named(
+   chr.pos.subject, chr.pos.pattern))

     chrom   chromStart    chromEnd
[1,] "chr10" "213,054,000" "213,055,000"
[2,] "chrM"  "111,000"     ""
[3,] NA      NA            NA
[4,] NA      NA            NA
[5,] "chr1"  "110"         "111"

>
```

Note that the third argument (list of conversion functions) is omitted in the code above. In that case, the return value is a character matrix, in which missing values indicate missing subjects or no match. The empty string is used for optional groups which are not used in the match (e.g. chromEnd group/column for second subject).

However we often want to extract numeric data; in this case we want to convert chromStart/End to integers. You can do that by supplying a named list of conversion functions as the third argument. Each function should take exactly one argument, a character vector (data in the matched column/group), and return a vector of the same size. The code below specifies the `int.from.digits` function for both chromStart and chromEnd:

```
> int.from.digits <- function(captured.text){
+   as.integer(gsub("[^0-9]", "", captured.text))
+ }
> conversion.list <- list(
+   chromStart=int.from.digits,
+   chromEnd=int.from.digits)
> (match.df <- namedCapture::str_match_named(
+   chr.pos.subject, chr.pos.pattern, conversion.list))

  chrom chromStart  chromEnd
1 chr10  213054000 213055000
2  chrM     111000        NA
3  <NA>         NA        NA
4  <NA>         NA        NA
5  chr1        110       111

>
```

Note that a data.frame is returned when the third argument is specified, in order to handle non-character data types returned by the conversion functions.

In the examples above the last subject has two possible matches, but only the first is returned by `str_match_named`. Use `str_match_all_named` to get all matches in each subject (not just the first match).

```
> namedCapture::str_match_all_named(
+   chr.pos.subject, chr.pos.pattern, conversion.list)

[[1]]
  chrom chromStart  chromEnd
1 chr10  213054000 213055000

[[2]]
  chrom chromStart chromEnd
1  chrM     111000       NA

[[3]]
data frame with 0 columns and 0 rows

[[4]]
data frame with 0 columns and 0 rows

[[5]]
  chrom chromStart chromEnd
1  chr1        110      111
2  chr2        220      222

>
```

As shown above, the result is a list with one element for each subject. Each list element is a data.frame with one row for each match.

Named output for named subjects

If the subject is named, its names will be used to name the output (rownames or list names).

```
> named.subject <- c(
+   ten="chr10:213,054,000-213,055,000",
+   M="chrM:111,000",
+   two="chr1:110-111 chr2:220-222") # two possible matches.
> namedCapture::str_match_named(
+   named.subject, chr.pos.pattern, conversion.list)

    chrom chromStart  chromEnd
ten chr10  213054000 213055000
M    chrM     111000        NA
two  chr1        110       111
```

```
> namedCapture::str_match_all_named(
+   named.subject, chr.pos.pattern, conversion.list)

$ten
  chrom chromStart  chromEnd
1 chr10  213054000 213055000

$M
  chrom chromStart chromEnd
1  chrM     111000       NA

$two
  chrom chromStart chromEnd
1  chr1        110      111
2  chr2        220      222

>
```

This feature makes it easy to select particular subjects/matches by name.

The `name` group specifies row names of output

If the pattern specifies the `name` group, then it will be used for the rownames of the output, and it will not be included as a column. However if the subject has names, and the `name` group is specified, then to avoid losing information the subject names are used to name the output (and the `name` column is included in the output).

```
> name.pattern <- paste0(
+   "(?P<name>chr.*?)",
+   ":",
+   "(?P<chromStart>[0-9,]+)",
+   "(?:",
+     "-",
+     "(?P<chromEnd>[0-9,]+)",
+   ")?")
> namedCapture::str_match_named(
+   named.subject, name.pattern, conversion.list)

     name chromStart  chromEnd
ten chr10  213054000 213055000
M    chrM     111000        NA
two  chr1        110       111

> namedCapture::str_match_all_named(
+   named.subject, name.pattern, conversion.list)

$ten
      chromStart  chromEnd
chr10  213054000 213055000

$M
     chromStart chromEnd
chrM     111000       NA

$two
     chromStart chromEnd
chr1        110      111
chr2        220      222

>
```

Readable and efficient variable argument syntax used in `str_match_variable`

In this section we introduce the variable argument syntax used in the `*_variable` functions. This new syntax is both readable and efficient, because it is motivated by the desire to avoid repetitive/boilerplate code. In the previous sections we defined the pattern using the `paste0` boilerplate, which is used to break the pattern over several lines for clarity. We begin by introducing str_match_variable, which extracts the first match from each subject. Using the variable argument syntax, we can omit paste0, and simply supply the pattern strings to str_match_variable directly,

```
> namedCapture::str_match_variable(
+   named.subject,
+   "(?P<chrom>chr.*?)",
+   ":",
+   "(?P<chromStart>[0-9,]+)",
+   "(?:",
+     "-",
+     "(?P<chromEnd>[0-9,]+)",
+   ")?")

    chrom   chromStart    chromEnd
ten "chr10" "213,054,000" "213,055,000"
M   "chrM"  "111,000"     ""
two "chr1"  "110"         "111"

>
```

The variable argument syntax allows further simplification by removing the named capture groups from the strings, and adding names to the corresponding arguments. For name1="pattern1", namedCapture internally generates/uses the regex (?P<name1>pattern1).

```
> namedCapture::str_match_variable(
+   named.subject,
+   chrom="chr.*?",
+   ":",
+   chromStart="[0-9,]+",
+   "(?:",
+     "-",
+     chromEnd="[0-9,]+",
+   ")?")

    chrom   chromStart    chromEnd
ten "chr10" "213,054,000" "213,055,000"
M   "chrM"  "111,000"     ""
two "chr1"  "110"         "111"

>
```

We can also provide a type conversion function on the same line as a named group:

```
> namedCapture::str_match_variable(
+   named.subject,
+   chrom="chr.*?",
+   ":",
+   chromStart="[0-9,]+", int.from.digits,
+   "(?:",
+     "-",
+     chromEnd="[0-9,]+", int.from.digits,
+   ")?")

    chrom chromStart  chromEnd
ten chr10  213054000 213055000
M    chrM     111000        NA
two  chr1        110       111

>
```

Note the repetition in the chromStart/End lines — the same pattern and type conversion function is used for each group. This repetition can be avoided by creating and using a sub-pattern list variable,

```
> int.pattern <- list("[0-9,]+", int.from.digits)
> namedCapture::str_match_variable(
+   named.subject,
+   chrom="chr.*?",
```

```
+   ":",
+   chromStart=int.pattern,
+   "(?:",
+     "-",
+     chromEnd=int.pattern,
+   ")?")

    chrom chromStart  chromEnd
ten chr10  213054000 213055000
M    chrM     111000        NA
two  chr1        110       111

>
```

Finally, the non-capturing group can be replaced by an un-named list:

```
> namedCapture::str_match_variable(
+   named.subject,
+   chrom="chr.*?",
+   ":",
+   chromStart=int.pattern,
+   list(
+     "-",
+     chromEnd=int.pattern
+   ), "?")

    chrom chromStart  chromEnd
ten chr10  213054000 213055000
M    chrM     111000        NA
two  chr1        110       111

>
```

In summary, the `str_match_variable` function takes a variable number of arguments, and allows for a shorter, less repetitive, and thus more user-friendly syntax:

- The first argument is the subject character vector.
- The other arguments specify the pattern, via character strings, functions, and/or lists.
- If a pattern (character/list) is named, we use the argument name in R for the capture group name in the regex.
- Each function is used to convert the text extracted by the previous named pattern argument. (type conversion can only be used with named R arguments, NOT with explicitly specified named groups in regex strings)
- R sub-pattern variables may be used to avoid repetition in the definition of the pattern and type conversion functions.
- Each list generates a group in the regex (named list = named capture group, un-named list = non-capturing group).
- All patterns are pasted together in the order that they appear in the argument list.

Extract all matches from a multi-line text file via `str_match_all_variable`

The variable argument syntax can also be used with `str_match_all_variable`, which is for the common case of extracting each match from a multi-line text file. In this section we demonstrate how to use `str_match_all_variable` to extract data.frames from a non-tabular text file.

```
> trackDb.txt.gz <- system.file(
+   "extdata", "trackDb.txt.gz", package="namedCapture")
> trackDb.lines <- readLines(trackDb.txt.gz)
>
```

Some representative lines from that file are shown below.

```
> cat(trackDb.lines[78:107], sep="\n")
```

```
track peaks_summary
type bigBed 5
shortLabel _model_peaks_summary
longLabel Regions with a peak in at least one sample
visibility pack
itemRgb off
spectrum on
bigDataUrl http://hubs.hpc.mcgill.ca/~thocking/PeakSegFPOP-/peaks_summary.bigBed


 track bcell_McGill0091
 parent bcell
 container multiWig
 type bigWig
 shortLabel bcell_McGill0091
 longLabel bcell | McGill0091
 graphType points
 aggregate transparentOverlay
 showSubtrackColorOnUi on
 maxHeightPixels 25:12:8
 visibility full
 autoScale on

  track bcell_McGill0091Coverage
  bigDataUrl http://hubs.hpc.mcgill.ca/~thocking/PeakSegFPOP-/samples/bcell/McGill0091/coverage.bigWig
  shortLabel bcell_McGill0091Coverage
  longLabel bcell | McGill0091 | Coverage
  parent bcell_McGill0091
  type bigWig
  color 141,211,199

>
```

Each block of text begins with track and includes several lines of data before the block ends with two consecutive newlines. That pattern is coded below:

```
> fields.mat <- namedCapture::str_match_all_variable(
+   trackDb.lines,
+   "track ",
+   name="\\S+",
+   fields="(?:\n[^\n]+)*",
+   "\n")
> head(substr(fields.mat, 1, 50))

                       fields
bcell                  "\nsuperTrack on show\nshortLabel bcell\nlongLabel bce"
kidneyCancer           "\nsuperTrack on show\nshortLabel kidneyCancer\nlongLa"
kidney                 "\nsuperTrack on show\nshortLabel kidney\nlongLabel ki"
leukemiaCD19CD10BCells "\nsuperTrack on show\nshortLabel leukemiaCD19CD10BCe"
monocyte               "\nsuperTrack on show\nshortLabel monocyte\nlongLabel "
skeletalMuscleCtrl     "\nsuperTrack on show\nshortLabel skeletalMuscleCtrl\n"

>
```

Note that this function assumes that its subject is a character vector with one element for each line in a file. The elements are pasted together using newline as a separator, and the regex is used to find all matches in the resulting multi-line string. The code above creates a data frame with one row for each track block, with rownames given by the track line (because of the capture group named name), and one fields column which is a string with the rest of the data in that block.

Each block has a variable number of lines/fields. Each line starts with a field name, followed by a space, followed by the field value. That regex is coded below:

```
> fields.list <- namedCapture::str_match_all_named(
+   fields.mat[, "fields"], paste0(
+     "\\s+",
+     "(?P<name>.*?)",
```

```
+     " ",
+     "(?P<value>[^\n]+)"))
> fields.list$bcell_McGill0091Coverage

           value
bigDataUrl "http://hubs.hpc.mcgill.ca/~thocking/PeakSegFPOP-/samples/bcell/McGill0091/coverage.bigWig"
shortLabel "bcell_McGill0091Coverage"
longLabel  "bcell | McGill0091 | Coverage"
parent     "bcell_McGill0091"
type       "bigWig"
color      "141,211,199"

>
```

The result is a list of data frames. There is a list element for each block, named by track. Each list element is a data frame with one row per field defined in that block (rownames are field names). The names/rownames make it easy to write R code that selects individual elements by name.

In the example above we extracted all fields from all tracks (using two regexes, one for the track, one for the field). In the example below we use a single regex to extract the name of each track, and split components into separate columns. It also demonstrates how to use nested named capture groups, via a named list which contains other named patterns.

```
> match.df <- namedCapture::str_match_all_variable(
+   trackDb.lines,
+   "track ",
+   name=list(
+     cellType=".*?",
+     "_",
+     sampleName=list(
+       "McGill",
+       sampleID=int.pattern),
+     dataType="Coverage|Peaks",
+     "|",
+     "[^\n]+"))
> match.df["bcell_McGill0091Coverage", ]

                         cellType sampleName sampleID dataType
bcell_McGill0091Coverage    bcell McGill0091       91 Coverage

>
```

Exercise for the reader: modify the above in order to capture the bigDataUrl field, and three additional columns (red, green, blue) from the color field.

`df_match_variable` extracts new columns from character columns in a data.frame

We also provide `namedCapture::df_match_variable` which extracts text from several columns of a data.frame, using a different named capture regular expression for each column.

- It requires a data.frame as the first argument.
- It takes a variable number of other arguments, all of which must be named. For each other argument we call `str_match_variable` on one column of the input data.frame.
- Each argument name specifies a column of the data.frame which will be used as the subject in `str_match_variable`.
- Each argument value specifies a pattern, in list/character/function variable argument syntax.
- The return value is a data.frame with the same number of rows as the input, but with an additional column for each named capture group. New columns are named using the convention `subjectColumnName.groupName`.

This function can greatly simplify the code required to create numeric data columns from character data columns. For example consider the following data which was output from the SLURM sacct command line program.

```
> (sacct.df <- data.frame(
+   Elapsed=c("07:04:42", "07:04:42", "07:04:49", "00:00:00", "00:00:00"),
+   JobID=c("13937810_25", "13937810_25.batch", "13937810_25.extern",
+     "14022192_[1-3]", "14022204_[4]"), stringsAsFactors=FALSE))

   Elapsed              JobID
1 07:04:42        13937810_25
2 07:04:42  13937810_25.batch
3 07:04:49 13937810_25.extern
4 00:00:00     14022192_[1-3]
5 00:00:00       14022204_[4]

>
```

Say we want to filter by the total Elapsed time (which is reported as hours:minutes:seconds), and base job id (which is the number before the underscore in the JobID column). We begin by defining a pattern that matches a range of integer task IDs in square brackets, and applying that pattern to the JobID column:

```
> range.pattern <- list(
+   "[[]",
+   task1=int.pattern,
+   list(
+     "-",
+     taskN=int.pattern
+   ), "?",
+   "[]]")
> namedCapture::df_match_variable(sacct.df, JobID=range.pattern)

   Elapsed              JobID JobID.task1 JobID.taskN
1 07:04:42        13937810_25          NA          NA
2 07:04:42  13937810_25.batch          NA          NA
3 07:04:49 13937810_25.extern          NA          NA
4 00:00:00     14022192_[1-3]           1           3
5 00:00:00       14022204_[4]           4          NA

>
```

The result shown above is another data frame with an additional column for each named capture group. Next, we define another pattern that matches either one task ID or the previously defined range pattern:

```
> task.pattern <- list(
+   "_", list(
+     task=int.pattern,
+     "|",#either one task(above) or range(below)
+     range.pattern))
> namedCapture::df_match_variable(sacct.df, JobID=task.pattern)

   Elapsed              JobID JobID.task JobID.task1 JobID.taskN
1 07:04:42        13937810_25         25          NA          NA
2 07:04:42  13937810_25.batch         25          NA          NA
3 07:04:49 13937810_25.extern         25          NA          NA
4 00:00:00     14022192_[1-3]         NA           1           3
5 00:00:00       14022204_[4]         NA           4          NA

>
```

Finally, we use the previously defined patterns to match the complete JobID column, along with the Elapsed column:

```
> namedCapture::df_match_variable(
+   sacct.df,
+   JobID=list(
+     job=int.pattern,
+     task.pattern,
+     list(
```

```
+       "[.]",
+       type=".*"
+     ), "?"),
+   Elapsed=list(
+     hours=int.pattern,
+     ":",
+     minutes=int.pattern,
+     ":",
+     seconds=int.pattern))

   Elapsed              JobID JobID.job JobID.task JobID.task1 JobID.taskN
1 07:04:42        13937810_25  13937810         25          NA          NA
2 07:04:42  13937810_25.batch  13937810         25          NA          NA
3 07:04:49 13937810_25.extern  13937810         25          NA          NA
4 00:00:00      14022192_[1-3]  14022192         NA           1           3
5 00:00:00        14022204_[4]  14022204         NA           4          NA
  JobID.type Elapsed.hours Elapsed.minutes Elapsed.seconds
1                        7               4              42
2      batch             7               4              42
3     extern             7               4              49
4                        0               0               0
5                        0               0               0

>
```

The code above specifies two named arguments to `df_match_variable`. Each named argument specifies a column from which tabular data are extracted using the corresponding pattern. The final result is a data frame with additional columns for each named capture group.

Comparisons with other R packages

In this section we compare the proposed functions in the namedCapture package with similar functions in other R packages for regular expressions.

Comparing namedCapture variable argument syntax with rex

In this section we compare namedCapture verbose variable argument syntax with the similar rex package. We have adapted the log parsing example from the rex package:

```
> log.subject <- 'gate3.fmr.com - - [05/Jul/1995:13:51:39 -0400] "GET /shuttle/countdown/
+ curly02.slip.yorku.ca - - [10/Jul/1995:23:11:49 -0400] "GET /sts-70/sts-70-patch-small.gif
+ boson.epita.fr - - [15/Jul/1995:11:27:49 -0400] "GET /movies/sts-71-mir-dock.MPG
+ 134.153.50.9 - - [13/Jul/1995:11:02:50 -0400] "GET /icons/text.xbm'
> log.lines <- strsplit(log.subject, split="\n")[[1]]
>
```

The goal is to extract the time and filetype for each log line. The code below uses the `rex` function to define a pattern for matching the filetype:

```
> library(rex)
> library(dplyr)
> (rex.filetype.pattern <- rex(
+   non_spaces, ".",
+   capture(name = 'filetype',
+           none_of(space, ".", "?", double_quote) %>% one_or_more())))

[^[:space:]]+\.(?<filetype>(?:[^[:space:].?"])+)

>
```

Note that rex defines R functions (e.g. `capture`, `one_or_more`) and constants (`non_spaces`, `double_quote`) which are translated to standard regular expression syntax via the `rex` function. These regex objects can be used as sub-patterns in other calls to `rex`, as in the code below:

```
> rex.pattern <- rex(
+   "[",
+   capture(name = "time", none_of("]") %>% zero_or_more()),
+   "]",
+   space, double_quote, "GET", space,
+   maybe(rex.filetype.pattern))
>
```

Finally, the pattern is used with `re_matches` in order to extract a data table, and the `mutate` function is used for type conversion:

```
> re_matches(log.lines, rex.pattern) %>% mutate(
+   filetype = tolower(filetype),
+   time = as.POSIXct(time, format="%d/%b/%Y:%H:%M:%S %z"))

                 time filetype
1 1995-07-05 10:51:39
2 1995-07-10 20:11:49      gif
3 1995-07-15 08:27:49      mpg
4 1995-07-13 08:02:50      xbm

>
```

Using the **namedCapture** package we begin by defining an analogous filetype pattern as a list containing literal regex strings and a type conversion function:

```
> namedCapture.filetype.pattern <- list(
+   "[^[:space:]]+[.]",
+   filetype='[^[:space:].?"]+', tolower)
>
```

We can then use that as a sub-pattern in a call to `str_match_variable`, which results in a data table with columns generated via the specified type conversion functions:

```
> namedCapture::str_match_variable(
+   log.lines,
+   "\\[",
+   time="[^]]*", function(x)as.POSIXct(x, format="%d/%b/%Y:%H:%M:%S %z"),
+   "\\]",
+   ' "GET ',
+   namedCapture.filetype.pattern, "?")

                 time filetype
1 1995-07-05 10:51:39
2 1995-07-10 20:11:49      gif
3 1995-07-15 08:27:49      mpg
4 1995-07-13 08:02:50      xbm

>
```

Overall both **rex** and **namedCapture** provide good support for defining regular expresions using a verbose, readable, and thus user-friendly syntax. However there are two major differences:

- **namedCapture** assumes the user knows regular expressions and can write them as R string literals; **rex** assumes the user knows its functions, which generate regex strings. For example the capture group time, `none_of("]") %>% zero_or_more()` in rex gets translated to the regex string `[^]]*`. Thus **rex** code is a bit more verbose than **namedCapture**.
- In **namedCapture** type conversion functions can be specified on the same line as the capture group name/pattern, whereas in **rex** type conversions are specified as a post-processing step on the result of `re_matches`.

Comparing `namedCapture::df_match_variable` with `tidyr::extract`

The **tidyr** package provides functionality similar to `namedCapture::df_match_variable`, which was introduced in Section "`df_match_variable` extracts new columns from character columns in a data.frame." Below we show how `tidyr::extract` can be used to compute a similar result as in that previous section, using the same data from the SLURM sacct command line program. We begin by defining a pattern which matches a range of integers in square brackets:

```
> tidyr.range.pattern <- "\\[([0-9]+)(?:-([0-9]+))?\\]"
> tidyr::extract(
+   sacct.df, "JobID", c("task1", "taskN"),
+   tidyr.range.pattern, remove=FALSE)
```

```
   Elapsed              JobID task1 taskN
1 07:04:42        13937810_25  <NA>  <NA>
2 07:04:42  13937810_25.batch  <NA>  <NA>
3 07:04:49 13937810_25.extern  <NA>  <NA>
4 00:00:00     14022192_[1-3]     1     3
5 00:00:00       14022204_[4]     4  <NA>
```

```
>
```

Note the pattern string includes un-named capture groups, because named capture is not supported. Names must therefore be specified in the third argument of `extract`. Next, we define a pattern which matches either a single task ID, or a range in square brackets:

```
> tidyr.task.pattern <- paste0(
+   "_(?:([0-9]+)|",
+   tidyr.range.pattern,
+   ")")
> tidyr::extract(
+   sacct.df, "JobID", c("task", "task1", "taskN"),
+   tidyr.task.pattern, remove=FALSE)
```

```
   Elapsed              JobID task task1 taskN
1 07:04:42        13937810_25   25  <NA>  <NA>
2 07:04:42  13937810_25.batch   25  <NA>  <NA>
3 07:04:49 13937810_25.extern   25  <NA>  <NA>
4 00:00:00     14022192_[1-3] <NA>     1     3
5 00:00:00       14022204_[4] <NA>     4  <NA>
```

```
>
```

In the code below we define a pattern that matches the entire job string:

```
> tidyr.job.pattern <- paste0(
+   "([0-9]+)",
+   tidyr.task.pattern,
+   "(?:[.](.*))?")
> (job.df <- tidyr::extract(
+   sacct.df, "JobID",
+   c("job", "task", "task1", "taskN", "type"),
+   tidyr.job.pattern))
```

```
   Elapsed      job task task1 taskN   type
1 07:04:42 13937810   25  <NA>  <NA>   <NA>
2 07:04:42 13937810   25  <NA>  <NA>  batch
3 07:04:49 13937810   25  <NA>  <NA> extern
4 00:00:00 14022192 <NA>     1     3   <NA>
5 00:00:00 14022204 <NA>     4  <NA>   <NA>
```

```
>
```

Finally, we use another pattern to extract the components of the elapsed time. Note that `convert=TRUE` means to use `utils::type.convert` on the result of each extracted group.

```
> tidyr::extract(
+   job.df, "Elapsed", c("hours", "minutes", "seconds"),
+   "([0-9]+):([0-9]+):([0-9]+)",
+   convert=TRUE)
```

```
  hours minutes seconds      job task task1 taskN   type
1     7       4      42 13937810   25  <NA>  <NA>   <NA>
2     7       4      42 13937810   25  <NA>  <NA>  batch
3     7       4      49 13937810   25  <NA>  <NA> extern
4     0       0       0 14022192 <NA>     1     3   <NA>
5     0       0       0 14022204 <NA>     4  <NA>   <NA>
```

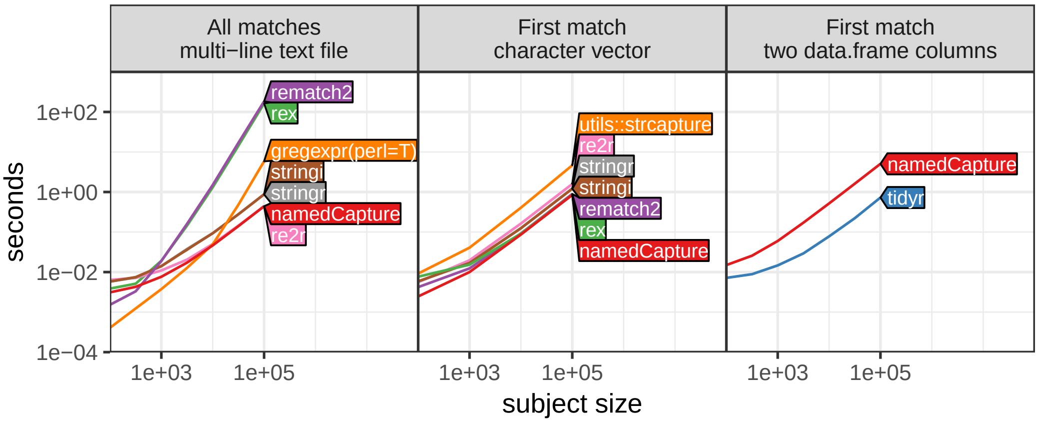

Figure 1: Computation time is plotted as a function of subject size (median line and quartile bands over 5 timings). Such timings are typical for real-world subjects and patterns such as the three examples shown.

>

Overall `tidyr::extract` functions similarly to `namedCapture::df_match_variable`, with the following differences:

- Because `tidyr::extract` uses the ICU C library, which does not support named capture regular expressions, it requires specifying the group names in a separate argument. In contrast, the **namedCapture** variable argument syntax supports specifying capture group names via R argument names on the same line as the corresponding sub-pattern.
- Whereas `tidyr::extract(convert=TRUE)` always uses `utils::type.convert` for type conversion, `namedCapture::df_match_variable` supports arbitrary group-specific type conversion functions, which are specified on the same line as the corresponding name/pattern.
- Because one call to `tidyr::extract` extracts data from one column in the subject data frame, it must be called twice (once for the Elapsed column, once for the JobID column). In contrast, one call to `namedCapture::df_match_variable` can be used to extract data from multiple columns in the subject data frame.

Comparing computation times of R regex packages

In this section we compare the computation time of the proposed **namedCapture** package with other R packages. For all of the comparisons, we used the **microbenchmark** package to compute the computation times of each R package/function. We study how the empirical computation time scales as a function of subject size. The first three comparisons come from the real-world examples discussed earlier in this article; the last two comparisons are pathological examples used to show worst case time complexity.

The first example involves extracting all matches from a multi-line text file, as discussed in Section "Extract all matches from a multi-line text file via `str_match_all_variable`." Figure 1 (left) shows comparisons with packages **re2r**, **stringr**, **stringi**, **rematch2**, **rex**. We expected small differences between the packages, on the order of constant factors. Surprisingly, the lines for the **rex** and **rematch2** packages have larger slopes than the other algorithms, which suggests quadratic rather than the expected linear time complexity. This can be explained because these packages use the base `gregexpr` and `substring` functions, which are implemented using inefficient quadratic time algorithms in R-3.5.2 (a bug report was posted on R-devel as a result of this investigation, and a fix should appear in a future version of R). **namedCapture** shows timings which are linear in the number of lines in the text file, similar to packages **stringr**, **stringi**, and **re2r**.

The second example involves extracting the first match from each line of a log file, as discussed in Section "Comparing **namedCapture** variable argument syntax with **rex**." Figure 1 (middle) shows comparisons with the previously discussed packages and `utils:strcapture`. We expected small differences between the packages, on the order of constant factors. In this comparison we observed only small constant factor differences, and linear time complexity for all packages.

The third example involves using a different regular expression to extract data for each of two columns of a data frame, as discussed in Section "Comparing `namedCapture::df_match_variable` with `tidyr::extract`." Figure 1 (right) shows a comparison with **tidyr**. Again we expected small

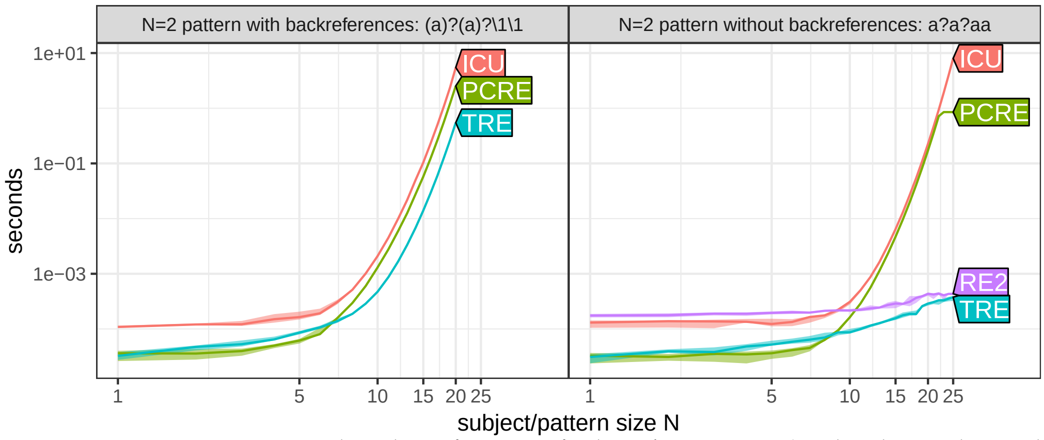

Figure 2: Computation time is plotted as a function of subject/pattern size (median line and quartile bands over 10 timings). For $N = 2$ the subject is aa and the pattern is shown in the facet title. Such slow timings only result from pathological subject/pattern combinations.

differences between the packages, and we observed linear time complexity for both **tidyr** and **namedCapture**.

The fourth example involves using a pathological regular expression of increasing size (with backreferences) on a subject of increasing size. Figure 2 (left) shows a comparison between ICU, PCRE, and TRE (RE2 is not included because it does not support backreferences). It is clear that all three libraries suffer from exponential time complexity. Although these timings are not typical, they illustrate the worst case time complexity that can be achieved. Such information should be considered along with other features (Table 1) when choosing a regex library. For example, guaranteed linear time complexity is essential for avoiding denial-of-service attacks in situations where potentially malicious users are permitted to define the regular expression pattern.

The final example involves using a pathological regular expression of increasing size (without backreferences) on a subject of increasing size. Figure 2 (right) shows a comparison between the previous libraries and additionally RE2. Is is clear that the fastest libraries are TRE and RE2, which exhibit linear time complexity. The slowest algorithm is clearly ICU, which exhibits exponential time complexity. The PCRE library is exponential up to a certain pattern/subject size, after which it is constant, because of a limit PCRE imposes on backtracking. Overall this comparison suggests that for guaranteed fast matching, RE2 must be used, via the **re2r** or **namedCapture** packages.

Discussion and conclusions

Our comparisons showed how similar operations can be performed by **namedCapture** and other R packages (e.g. **tidyr** and **rex**). Our empirical timings highlight the relative advantages and disadvantages of the different R packages we tested. For example, we observed that **rex** and **rematch2** are relatively slow for finding all matches in large multi-line text files, because they use the base `gregexpr`/`substring` functions (which use inefficient quadratic time algorithms in R-3.5.2 but hopefully will be linear time in a future version of R). In contrast, **namedCapture** showed relatively fast empirical timings, which were within a constant factor of other packages (**stringi**, **stringr**, **re2r**).

The article presented the **namedCapture** package, along with detailed comparisons with other R packages for regular expressions. A unique feature of the **namedCapture** package is its compact and readable syntax for defining regular expressions in R code. We showed how this syntax can be used to extract data tables from a variety of non-tabular text data. We also highlighted several other features of the **namedCapture** package, which include support for arbitrary type conversion functions, named output based on subject names and the `name` capture group, and two regex engines (PCRE and RE2). PCRE can be used for backreferences (e.g. for matching HTML tags), but otherwise RE2 should be preferred for guaranteed linear time complexity. Overall we hope that the unique features of the **namedCapture** package will be useful and inspiring for other package developers.

Reproducible research statement. The source code for this article can be freely downloaded from `https://github.com/tdhock/namedCapture-article`

Bibliography

G. Csárdi. *rematch2: Tidy Output from Regular Expression Matching*, 2017. URL https://CRAN.R-project.org/package=rematch2. R package version 2.0.1. [p2]

J. E. F. Friedl. *Mastering Regular Expressions*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2 edition, 2002. [p1]

M. Gagolewski. *R package stringi: Character string processing facilities*, 2018. URL http://www.gagolewski.com/software/stringi/. [p2]

P. Hazel. ChangeLog for PCRE, 2003. URL https://github.com/tdhock/regex-tutorial/blob/master/pcre1-changelog.txt. [p1]

T. D. Hocking. Bug 14518 - wishlist: named capture in regular expressions, 2011a. URL https://bugs.r-project.org/bugzilla3/show_bug.cgi?id=14518. [p2]

T. D. Hocking. Fast, named capture regular expressions in R 2.14. In *useR 2011 conference proceedings*, 2011b. URL http://web.warwick.ac.uk/statsdept/user-2011/TalkSlides/Lightening/2-StatisticsAndProg_3-Hocking.pdf. [p2]

W. L. Johnson, J. H. Porter, S. I. Ackley, and D. T. Ross. Automatic generation of efficient lexical processors using finite state techniques. *Commun. ACM*, 11(12):805–813, Dec. 1968. ISSN 0001-0782. doi: 10.1145/364175.364185. [p1]

S. C. Kleene. Representation of events in nerve nets and finite automata. In C. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–41. Princeton University Press, Princeton, NJ, 1956. URL http://www.diku.dk/hjemmesider/ansatte/henglein/papers/kleene1956.pdf. [p1]

V. Laurikari. *TRE: The free and portable approximate regex matching library*, 2019. URL https://laurikari.net/tre/. [p1]

Python developers. Python 1.5.2 history, 1997a. URL https://github.com/tdhock/regex-tutorial/blob/master/python-1.5.2-Misc-HISTORY.txt. [p1]

Python developers. Python documentation for built-in module re, 1997b. URL https://github.com/tdhock/regex-tutorial/blob/master/python-1.5-Doc-libre.tex. [p1]

Qin Wenfeng. *re2r: RE2 Regular Expression*, 2017. URL https://CRAN.R-project.org/package=re2r. R package version 0.2.0. [p2]

R Core Team. News for the 1.x series, 2002. URL https://github.com/tdhock/regex-tutorial/blob/master/R.NEWS.1.txt. [p1]

K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6): 419–422, June 1968. ISSN 0001-0782. [p1]

K. Ushey, J. Hester, and R. Krzyzanowski. *rex: Friendly Regular Expressions*, 2017. URL https://CRAN.R-project.org/package=rex. R package version 1.1.2. [p2]

H. Wickham and L. Henry. *tidyr: Easily Tidy Data with 'spread()' and 'gather()' Functions*, 2018. URL https://CRAN.R-project.org/package=tidyr. R package version 0.8.2. [p2]

Toby Dylan Hocking
School of Informatics, Computing, and Cyber Systems
Northern Arizona University
Flagstaff, Arizona
USA
toby.hocking@nau.edu