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).

In this article our original contribution is the R package namedCapture which provides several new features for named capture regular expressions. The main new ideas are (1) group-specific type conversion functions, (2) a user-friendly syntax for defining group names with R argument names, and (3) named output based on subject names and the name capture group.

The organization of this article is as follows. The rest of this introduction provides a brief history of regular expressions and their usage in R, then gives an overview of current R packages for regular expressions. The second section describes the proposed functions of the namedCapture package. The third section provides detailed comparisons with other R packages, in terms of syntax and computation times. The article concludes with a summary and discussion.

Origin of regular expressions and named capture groups

Regular expressions were first proposed in a theoretical 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>...) syntax without a P, and the (?labelname ...) syntax with single quotes, to be consistent with Perl and .NET (Hazel, 2003).

In the R NEWS files, the first mention of regular expression support was in 1997 with R-0.60,
“Regular expression matching is now done with system versions of the regexp library” (R Core Team, 1997). Starting with R-0.99.0, “R now compiles in the GNU version of regexp” (R Core Team, 1997). PCRE was first included in R version 1.6.0 in 2002 (R Core Team, 2002). R-2.10 in 2009 was the first version to deprecate basic regular expressions, extended=FALSE, which are no longer supported (R Core Team, 2009). TRE is another C library for regular expressions that was included in R starting in R-2.10 (Laurikari, 2019). Although TRE supports capture groups, it does not allow capture groups to be named. 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 most important feature for the purposes of this paper is “Output group names” which means the C library supports specifying capture group names in the regular expression pattern via ⟨?<group>pattern⟩ or ⟨?P<group>pattern⟩, and then extracting those names for use in R (typically as column names in the resulting match matrix). “Worst case linear time” means that the match time is linear in the length of the input string, which is only guaranteed by the RE2 library. “Backreferences” can be used in patterns such as (.)\1, which means to match any character that appears twice in a row. “Atomic grouping” or “possessive quantifiers” means that only the greediest option of all possible alternatives will be considered; an example is the pattern ⟨?+.⟩+bar which does not match the subject foobar (whereas the analog without the atomic group does). “Unicode properties” means support for regular expressions such as \p{EMOJI_Presentation}, which only work with ICU. “Lookaround” means support for zero-length assertions such as foo(?=bar) which matches foo only when it is followed by bar (but bar is not included in the match). “Recursion” is useful for matching balanced parentheses, and is only supported in PCRE; a simple recursive pattern is ⟨?R⟩?z which matches one or more a followed by exactly the same number of z (e.g. asazzz).

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). The C code that uses PCRE to extract each named capture group was accepted into R starting with version 2.14 (Hocking, 2011a). A lightning talk at useR 2011 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). In this section we give an overview of the features of each package; in the section “Comparisons with other R packages” we show detailed comparisons including sample R code and outputs.

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 `bind_re_match` for matching data.frame columns, and `re_match_all` which extracts all matches using the base `gregexpr` function. All functions output a data.frame 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
Table 2: R packages that provide functions for extracting first/all regex matches, and C library used.

<table>
<thead>
<tr>
<th>Package</th>
<th>First match</th>
<th>All matches</th>
<th>C library</th>
</tr>
</thead>
<tbody>
<tr>
<td>namedCapture</td>
<td>*_match_variable</td>
<td>str_match_all_variable</td>
<td>PCRE/RE2</td>
</tr>
<tr>
<td>base</td>
<td>regexpr</td>
<td>gregexpr</td>
<td>PCRE/TRE</td>
</tr>
<tr>
<td>utils</td>
<td>strcapture</td>
<td>N/A</td>
<td>PCRE/TRE</td>
</tr>
<tr>
<td>rematch2</td>
<td>re_match, bind_re_match</td>
<td>re_match_all</td>
<td>PCRE/TRE</td>
</tr>
<tr>
<td>rex</td>
<td>re_matches(global=FALSE)</td>
<td>re_matches(global=TRUE)</td>
<td>PCRE</td>
</tr>
<tr>
<td>stringr</td>
<td>str_match</td>
<td>str_match_all</td>
<td>ICU</td>
</tr>
<tr>
<td>stringi</td>
<td>stri_match</td>
<td>stri_match_all</td>
<td>ICU</td>
</tr>
<tr>
<td>tidyr</td>
<td>extract</td>
<td>N/A</td>
<td>ICU</td>
</tr>
<tr>
<td>re2r</td>
<td>re2_match</td>
<td>re2_match_all</td>
<td>RE2</td>
</tr>
</tbody>
</table>

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 main design features of the namedCapture package are inspired by the base R system, which
Table 3: R packages provide different options for subject/pattern input, extracted text outputs, named capture groups output to R, and type conversion. Abbreviations: chr=character vector, df=data.frame, mat=character matrix, verbose=regex defined in R code which is translated to a character string, compiled=regex string may be compiled and saved to an R object for later use.

<table>
<thead>
<tr>
<th>Package/function</th>
<th>subject</th>
<th>pattern</th>
<th>outputs</th>
<th>named</th>
<th>types</th>
</tr>
</thead>
<tbody>
<tr>
<td>namedCapture</td>
<td>chr/df</td>
<td>chr/verbose</td>
<td>mat/list/df</td>
<td>yes</td>
<td>any</td>
</tr>
<tr>
<td>base</td>
<td>chr</td>
<td>chr</td>
<td>mat/list</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>utils::strcapture</td>
<td>chr</td>
<td>chr</td>
<td>df</td>
<td>no</td>
<td>some</td>
</tr>
<tr>
<td>rematch2</td>
<td>chr/df</td>
<td>chr</td>
<td>df</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>rex</td>
<td>chr</td>
<td>chr/verbose</td>
<td>df/list</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>stringr</td>
<td>chr</td>
<td>chr</td>
<td>mat/list</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>stringi</td>
<td>chr</td>
<td>chr</td>
<td>mat/list</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>tidyr::extract</td>
<td>df</td>
<td>chr</td>
<td>df</td>
<td>no</td>
<td>some</td>
</tr>
<tr>
<td>re2r</td>
<td>chr</td>
<td>chr/compiled</td>
<td>df/list</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

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, and are mostly for internal use; the *_variable functions take a variable number of arguments and are the recommended functions to use.

<table>
<thead>
<tr>
<th>function</th>
<th>subject</th>
<th>pattern</th>
<th>outputs</th>
<th>arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>str_match_named</td>
<td>chr</td>
<td>chr</td>
<td>df</td>
<td>chr subject, chr pattern, functions</td>
</tr>
<tr>
<td>str_match_all_named</td>
<td></td>
<td></td>
<td>df/list</td>
<td>chr subject, chr/list/function, ...</td>
</tr>
<tr>
<td>str_match_variable</td>
<td></td>
<td></td>
<td></td>
<td>df subject, chr/list/function, ...</td>
</tr>
<tr>
<td>df_match_variable</td>
<td>NA</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.
- 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.

Since introduction of the variable argument syntax (explained later in this section), these functions are mostly for internal use. Here we give an example of their usage, because it is similar to other R regex packages which some readers are probably already familiar with. Consider subjects containing genomic positions:

```r
> chr.pos.subject <- c("chr10:213,054,000-213,055,000", "chrM:111,000", "+ "this will not match", NA, "chr1:110-111 chr2:220-222")
```

These subjects consist of a chromosome name string, a colon, a start position, and optionally a dash and an end position. The following pattern is used to extract those data:
The pattern above is defined using `paste0`, writing each named capture group on a separate line, which increases readability of the pattern. Note that an optional non-capturing group begins with `(?:` and ends with `)`?. In the code below, we use the `str_match_named` function on the previously defined subject and pattern:

```r
> (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"
```

When the third argument is omitted, the return value is 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. 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). This output format is similar to the output of `stringi::stri_match` and `stringr::str_match`; these other functions also report a column for the entire match, whereas `namedCapture::str_match_named` only reports a column for each named capture group.

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:

```r
> 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)
> str(match.df)

'data.frame': 5 obs. of 3 variables:
$ chrom : chr "chr10" "chrM" NA NA ...
$ chromStart: int 213054000 111000 NA NA 110
$ chromEnd : int 213055000 NA NA NA 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).

```r
> 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]]
```
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).

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

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

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

$ten
  chrom chromStart chromEnd
ten 1 chr10 213054000 213055000

$M
  chrom chromStart chromEnd
  M 1 chrM 111000 NA

$two
  chrom chromStart chromEnd
two 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).

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

```r
name chromStart chromEnd
ten 1 chr10 213054000 213055000
```

---

The R Journal Vol. XX/YY, AAAA 20ZZ

ISSN 2073-4859
In this section we introduce the variable argument syntax used in the \_*variable\_functions, which is the recommended way to use \_namedCapture. 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 paste\_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 paste\_0, and simply supply the pattern strings to str\_match\_variable directly,

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

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).

```r
> namedCapture::str_match_variable(
+   named.subject,
+   "chr\:<\:\:\(\:\?\:\:\:\?\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\\```
> namedCapture::str_match_variable(
+   named.subject,
+   "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,
+   "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,
+   "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.

```r
> 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.

```r
> show.width <- 55
> substr(trackDb.lines[78:107], 1, show.width)
```

```
[1] "track peaks_summary"
[2] "type bigBed 5"
[3] "shortLabel _model_peaks_summary"
[4] "longLabel Regions with a peak in at least one sample"
[5] "visibility pack"
[6] "itemRgb off"
[7] "spectrum on"
[8] "bigDataUrl http://hubs.hpc.mcgill.ca/~thocking/PeakSegF"
[9] ""
[10] ""
[12] " parent bcell"
[13] " container multiWig"
[14] " type bigWig"
[16] " longLabel bcell | McGill0091"
[17] " graphType points"
[18] " aggregate transparentOverlay"
[19] " showSubtrackColorOnUi on"
[20] " maxHeightPixels 25:12:8"
[21] " visibility full"
[22] " autoScale on"
[23] ""
[24] " track bcell_McGill0091Coverage"
[26] " shortLabel bcell_McGill0091Coverage"
[27] " longLabel bcell | McGill0091 | Coverage"
[28] " parent bcell_McGill0091"
[29] " type bigWig"
[30] " 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:

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

```r
cell    "\nsuperTrack on show\shortLabel bcell\nlongLabel bcell Ch"  kidneyCancer    "\nsuperTrack on show\shortLabel kidneyCancer\nlongLabel k"
kidney    "\nsuperTrack on show\shortLabel kidney\nlongLabel kidney " leukemiaCD19CD10BCells  "\nsuperTrack on show\shortLabel leukemiaCD19CD10BCells\nlongLabel monocyte    "\nsuperTrack on show\shortLabel monocyte\nlongLabel monoch skeletalMuscleCtrl "\nsuperTrack on show\shortLabel skeletalMuscleCtrl}\nlongL"
```

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 name capture group), 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:

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

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.

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

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.

```r
(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))
```

<table>
<thead>
<tr>
<th>Elapsed</th>
<th>JobID</th>
</tr>
</thead>
<tbody>
<tr>
<td>07:04:42</td>
<td>13937810_25</td>
</tr>
<tr>
<td>07:04:42</td>
<td>13937810_25.batch</td>
</tr>
<tr>
<td>07:04:49</td>
<td>13937810_25.extern</td>
</tr>
<tr>
<td>00:00:00</td>
<td>14022192_[1-3]</td>
</tr>
<tr>
<td>00:00:00</td>
<td>14022204_[4]</td>
</tr>
</tbody>
</table>

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:

```r
range.pattern <- list("[]", task1=int.pattern, list("-", taskN=int.pattern), ");", "+ ")[")

> namedCapture::df_match_variable(sacct.df, JobID=range.pattern)
```

<table>
<thead>
<tr>
<th>Elapsed</th>
<th>JobID</th>
<th>JobID.task1</th>
<th>JobID.taskN</th>
</tr>
</thead>
<tbody>
<tr>
<td>07:04:42</td>
<td>13937810_25</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>07:04:42</td>
<td>13937810_25.batch</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>07:04:49</td>
<td>13937810_25.extern</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>00:00:00</td>
<td>14022192_[1-3]</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>00:00:00</td>
<td>14022204_[4]</td>
<td>4</td>
<td>NA</td>
</tr>
</tbody>
</table>

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:

```r
task.pattern <- list("_", list(task=int.pattern, "+ "[1], #either one task(above) or range(below) + range.pattern))
```

> namedCapture::df_match_variable(sacct.df, JobID=task.pattern)

<table>
<thead>
<tr>
<th>Elapsed</th>
<th>JobID</th>
<th>JobID.task</th>
<th>JobID.task1</th>
<th>JobID.taskN</th>
</tr>
</thead>
<tbody>
<tr>
<td>07:04:42</td>
<td>13937810_25</td>
<td>25</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>07:04:42</td>
<td>13937810_25.batch</td>
<td>25</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>07:04:49</td>
<td>13937810_25.extern</td>
<td>25</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>00:00:00</td>
<td>14022192_[1-3]</td>
<td>NA</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>00:00:00</td>
<td>14022204_[4]</td>
<td>NA</td>
<td>4</td>
<td>NA</td>
</tr>
</tbody>
</table>

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

```r
future::plan("multiprocess")
> namedCapture::df_match_variable(
+ sacct.df, 
+ JobID=list(
+ job=int.pattern, 
+ task.pattern, 
+ list(task1=int.pattern, 
+ taskN=int.pattern))
```

```r
Elapsed | JobID | JobID.task | JobID.task1 | JobID.taskN |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>07:04:42</td>
<td>13937810_25</td>
<td>25</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>07:04:42</td>
<td>13937810_25.batch</td>
<td>25</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>07:04:49</td>
<td>13937810_25.extern</td>
<td>25</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>00:00:00</td>
<td>14022192_[1-3]</td>
<td>NA</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>00:00:00</td>
<td>14022204_[4]</td>
<td>NA</td>
<td>4</td>
<td>NA</td>
</tr>
</tbody>
</table>
The code above specifies two named arguments to \texttt{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 an additional column for each named capture group.

**Comparisons with other R packages**

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

Comparing \texttt{namedCapture} variable argument syntax with rex

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

```r

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

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

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:

```r
> (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:

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

<table>
<thead>
<tr>
<th>time</th>
<th>filetype</th>
</tr>
</thead>
<tbody>
<tr>
<td>1995-07-05 10:51:39</td>
<td></td>
</tr>
<tr>
<td>1995-07-10 20:11:49</td>
<td>gif</td>
</tr>
<tr>
<td>1995-07-15 08:27:49</td>
<td>mpg</td>
</tr>
<tr>
<td>1995-07-13 08:02:50</td>
<td>xbm</td>
</tr>
</tbody>
</table>

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

```r
> namedCapture.filetype.pattern <- list( 
  + 
    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:

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

<table>
<thead>
<tr>
<th>time</th>
<th>filetype</th>
</tr>
</thead>
<tbody>
<tr>
<td>1995-07-05 10:51:39</td>
<td></td>
</tr>
<tr>
<td>1995-07-10 20:11:49</td>
<td>gif</td>
</tr>
<tr>
<td>1995-07-15 08:27:49</td>
<td>mpg</td>
</tr>
<tr>
<td>1995-07-13 08:02:50</td>
<td>xbm</td>
</tr>
</tbody>
</table>

Overall both `rex` and `namedCapture` provide good support for defining regular expressions 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("[\"]\")` in `rex` gets translated to the regex string `\[^[\[:space:]]+\]^`. 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 other functions for data.frames**

The `tidyr` and `rematch2` packages provide 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:

```r
> tidyr.range.pattern <- "\[[0-9]+(?:-([0-9]+))??\]
```

<table>
<thead>
<tr>
<th>Elapsed</th>
<th>JobID</th>
<th>task1</th>
<th>taskN</th>
</tr>
</thead>
<tbody>
<tr>
<td>07:04:42</td>
<td>13937810.25</td>
<td>&lt;NA&gt;</td>
<td>&lt;NA&gt;</td>
</tr>
<tr>
<td>07:04:42</td>
<td>13937810.25.batch</td>
<td>&lt;NA&gt;</td>
<td>&lt;NA&gt;</td>
</tr>
<tr>
<td>07:04:49</td>
<td>13937810.25.extern</td>
<td>&lt;NA&gt;</td>
<td>&lt;NA&gt;</td>
</tr>
<tr>
<td>00:00:00</td>
<td>14022192[1-3]</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>00:00:00</td>
<td>14022204[4]</td>
<td>4</td>
<td>&lt;NA&gt;</td>
</tr>
</tbody>
</table>
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:

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

<table>
<thead>
<tr>
<th>Elapsed</th>
<th>JobID</th>
<th>task</th>
<th>task1</th>
<th>taskN</th>
</tr>
</thead>
<tbody>
<tr>
<td>07:04:42</td>
<td>13937810_25</td>
<td>25</td>
<td>&lt;NA&gt;</td>
<td>&lt;NA&gt;</td>
</tr>
<tr>
<td>07:04:42</td>
<td>13937810_25.batch</td>
<td>25</td>
<td>&lt;NA&gt;</td>
<td>&lt;NA&gt;</td>
</tr>
<tr>
<td>07:04:49</td>
<td>13937810_25.extern</td>
<td>25</td>
<td>&lt;NA&gt;</td>
<td>&lt;NA&gt;</td>
</tr>
<tr>
<td>00:00:00</td>
<td>14022192_[1-3]</td>
<td>&lt;NA&gt;</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>00:00:00</td>
<td>14022204_[4]</td>
<td>&lt;NA&gt;</td>
<td>4</td>
<td>&lt;NA&gt;</td>
</tr>
</tbody>
</table>

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

```r
> 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))
```

<table>
<thead>
<tr>
<th>Elapsed</th>
<th>job</th>
<th>task</th>
<th>task1</th>
<th>taskN</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>07:04:42</td>
<td>13937810</td>
<td>25</td>
<td>&lt;NA&gt;</td>
<td>&lt;NA&gt;</td>
<td></td>
</tr>
<tr>
<td>07:04:42</td>
<td>13937810</td>
<td>25</td>
<td>&lt;NA&gt;</td>
<td>&lt;NA&gt;</td>
<td>batch</td>
</tr>
<tr>
<td>07:04:49</td>
<td>13937810</td>
<td>25</td>
<td>&lt;NA&gt;</td>
<td>&lt;NA&gt;</td>
<td>extern</td>
</tr>
<tr>
<td>00:00:00</td>
<td>14022192</td>
<td>&lt;NA&gt;</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>00:00:00</td>
<td>14022204</td>
<td>&lt;NA&gt;</td>
<td>4</td>
<td>&lt;NA&gt;</td>
<td></td>
</tr>
</tbody>
</table>

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.

```r
> tidyr::extract(job.df, "Elapsed", c("hours", "minutes", "seconds"),
```

<table>
<thead>
<tr>
<th>hours</th>
<th>minutes</th>
<th>seconds</th>
<th>job</th>
<th>task</th>
<th>task1</th>
<th>taskN</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>42</td>
<td>13937810</td>
<td>25</td>
<td>&lt;NA&gt;</td>
<td>&lt;NA&gt;</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>4</td>
<td>13937810</td>
<td>25</td>
<td>&lt;NA&gt;</td>
<td>&lt;NA&gt;</td>
<td>batch</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>4</td>
<td>13937810</td>
<td>25</td>
<td>&lt;NA&gt;</td>
<td>&lt;NA&gt;</td>
<td>extern</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>14022192</td>
<td>&lt;NA&gt;</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>14022204</td>
<td>&lt;NA&gt;</td>
<td>4</td>
<td>&lt;NA&gt;</td>
<td></td>
</tr>
</tbody>
</table>

Below we show the same computation using `rematch2::bind_re_match`, which supports named capture. Note that we use `paste0` to define a regular expression with each named capture group on a separate line:

```r
> rematch2.range.pattern <- paste0(
  + "\[",
  + "(?<task1>[0-9]+)",
  + ")\]
  + "(?<taskN>[0-9]+)",
  + ")\]"
> rematch2::bind_re_match(sacct.df, JobID, rematch2.range.pattern)
```

<table>
<thead>
<tr>
<th>Elapsed</th>
<th>JobID</th>
<th>task1</th>
<th>taskN</th>
</tr>
</thead>
<tbody>
<tr>
<td>07:04:42</td>
<td>13937810_25</td>
<td>&lt;NA&gt;</td>
<td>&lt;NA&gt;</td>
</tr>
<tr>
<td>07:04:42</td>
<td>13937810_25.batch</td>
<td>&lt;NA&gt;</td>
<td>&lt;NA&gt;</td>
</tr>
<tr>
<td>07:04:49</td>
<td>13937810_25.extern</td>
<td>&lt;NA&gt;</td>
<td>&lt;NA&gt;</td>
</tr>
<tr>
<td>00:00:00</td>
<td>14022192_[1-3]</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>00:00:00</td>
<td>14022204_[4]</td>
<td>4</td>
<td>&lt;NA&gt;</td>
</tr>
</tbody>
</table>

Above we extract a range of task IDs in square brackets, and below we optionally match a single task ID:

```r
> rematch2.task.pattern <- paste0(
  + "(?<task>[0-9]+)",
  + "([0-9]+)",
  + ")",
  + rematch2.range.pattern, ")")
> rematch2::bind_re_match(sacct.df, JobID, rematch2.task.pattern)
```
Below we additionally match the job ID and job type:

```r
> rematch2.job.pattern <- paste0(
+ "(?P<job>[0-9]+)",
+ rematch2.task.pattern,
+ "(:[.]",
+ "(?P<type>\.+)",
+ ")")
> (rematch2.job.df <- rematch2::bind_re_match(
+ sacct.df, JobID, rematch2.job.pattern))
```

<table>
<thead>
<tr>
<th>Elapsed</th>
<th>JobID</th>
<th>job</th>
<th>task</th>
<th>task1</th>
<th>taskN</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>07:04:42</td>
<td>13937810_25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>07:04:42</td>
<td>13937810_25.batch</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>batch</td>
</tr>
<tr>
<td>07:04:49</td>
<td>13937810_25.extern</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>extern</td>
</tr>
<tr>
<td>00:00:00</td>
<td>14022192_[1-3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 3</td>
</tr>
<tr>
<td>00:00:00</td>
<td>14022204_[4]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>

Finally we call the function on the result from above with a new pattern for another column:

```r
> transform(rematch2::bind_re_match(
+ rematch2.job.df, Elapsed,
+ "(?P<hours>[0-9]+):(?P<minutes>[0-9]+):(?P<seconds>[0-9]+)"),
+ hours.int=as.integer(hours),
+ minutes.int=as.integer(minutes),
+ seconds.int=as.integer(seconds))
```

<table>
<thead>
<tr>
<th>Elapsed</th>
<th>JobID</th>
<th>job</th>
<th>task</th>
<th>task1</th>
<th>taskN</th>
<th>type</th>
<th>hours</th>
<th>minutes</th>
<th>seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>07:04:42</td>
<td>13937810_25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>07</td>
<td>04</td>
<td>00</td>
</tr>
<tr>
<td>07:04:42</td>
<td>13937810_25.batch</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>batch</td>
<td>07</td>
<td>04</td>
<td>00</td>
</tr>
<tr>
<td>07:04:49</td>
<td>13937810_25.extern</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>extern</td>
<td>07</td>
<td>04</td>
<td>00</td>
</tr>
<tr>
<td>00:00:00</td>
<td>14022192_[1-3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 3</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00:00:00</td>
<td>14022204_[4]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

Overall our comparison demonstrates that `tidyr::extract` and `rematch2::bind_re_match` function 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, `rematch2` supports specifying capture group names in regex string literals; `namedCapture` variable argument syntax supports specifying capture group names as R argument names on the same line as the corresponding sub-pattern.
- Since `rematch2::bind_re_match` returns character columns, conversion to numeric types must be accomplished in a post-processing step using a function such as `transform`. In contrast `tidyr::extract(convert=TRUE)` always uses `utils::type.convert` for type conversion, and `namedCapture::df_match_variable` supports arbitrary group-specific type conversion functions, which are specified on the same line as the corresponding name/pattern.
- Because `tidyr::extract` and `rematch2::bind_re_match` operate on one column in the subject data frame, they 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/pattern 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 \texttt{str_match_all_variable}.” Figure 1 shows comparisons with packages \texttt{re2r}, \texttt{stringr}, \texttt{stringi}, \texttt{rematch2}, \texttt{rex}. We expected small differences between the packages, on the order of constant factors. Using R-3.5.2 (left panel of Figure 1), the lines for the \texttt{rex} and \texttt{rematch2} packages have significantly larger slopes than the other packages (\texttt{namedCapture}, \texttt{stringr}, \texttt{stringi}, and \texttt{re2r}). This can be explained because \texttt{rex} and \texttt{rematch2} use the base gregexpr and substring functions, which are implemented using inefficient quadratic time algorithms in R-3.5.2. As a result of this research, this issue was reported on the R-devel email list, and R-core member Tomas Kalibera has fixed the problem. In R-3.6.0 (right panel of Figure 1), linear time algorithms are used.

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 \texttt{rex}.” Figure 2 (left) shows comparisons with the previously discussed packages and \texttt{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 other functions for data.frames.” Figure 2 (right) shows a comparison with \texttt{tidyr} and \texttt{rematch2}. Again we expected small differences between the packages, and we observed linear time complexity for \texttt{tidyr}, \texttt{rematch2}, and \texttt{namedCapture} (using either PCRE or RE2).

The fourth example shows the worst case time complexity, using a pathological regular expression of increasing size (with backreferences) on a subject of increasing size. For example with size $N = 2$ we use the regex (a)?(a)?\1 on the pattern aa; the match time complexity is $O(2^N)$. Note that possessive quantifiers, (a)?+, could be used to avoid the exponential time complexity (but possessive quantifiers are only supported in PCRE and ICU, not TRE nor RE2). Figure 3 (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 3 (right) shows a comparison between the previous libraries and additionally RE2. It is clear that the fastest libraries are TRE and RE2, which...
Figure 2: Computation time for finding the first match is plotted as a function of subject size (median lines and quartile bands over 5 timings). Such timings are typical for real-world subjects and patterns such as the two examples shown.

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 default limit PCRE imposes on backtracking. If other libraries allow configuring a limit on backtracking, such an option could be used to avoid this exponential time complexity. Again these timings are on synthetic data which achieve the worst case time complexity, and are not typical of real data. 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 revealed an inefficient implementation of the substring/gregexpr functions in R-3.5.2, which was fixed in R-3.6.0 as a result of this research. After applying that fix, all packages were asymptotically linear in our empirical comparisons of time to compute matches using typical/real-world patterns and subjects. Finally, we studied the worst-case time complexity of matching on pathological patterns/subjects, and showed that RE2 must be used for guaranteed linear time complexity.

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. The ICU library may be preferred for its strong unicode support (Table 1), so we are considering implementing ICU as another regex engine usable in namedCapture.

We thank a reviewer for a suggestion about other choices for the variable argument syntax for specifying type conversion functions. The current syntax uses a named R argument to specify the capture group name, then a character string literal to specify the capture group pattern, then a function name specify the type conversion. Other choices could use formulas or the := operator to define type conversions. 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

Figure 3: Computation time is plotted as a function of subject/pattern size (median lines 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.


R Core Team. News for the 0.x series, 1997. URL https://cloud.r-project.org/src/base/NEWS.0. [p2]


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