String Processing Macros (`strings.dtx`)

Charles Duan

November 16, 2023

This package provides a variety of low-level text processing commands and macro helpers. The section in the main *Hereinafter* documentation just provides an overview of features; the full documentation for this file should be consulted for the particular macros and their usage.

Initially, the package provides several syntatic sugar macros for conditionals, definitions, and expansion control. These are similar to the `etoolbox` package.

```
\ProvidesPackage{strings}[2004/10/09 String tests]
\RequirePackage{etoolbox}
```

1 Conditionals

Performs an .-type test. Takes an argument of an if-conditional, like:

```
\ifx#1\test
```

and turns it into:

```
\ifx#1\test
    \expandafter\@firstoftwo
\else
    \expandafter\@secondoftwo
```

Usage is like:

```
\@test\if⟨test⟩\fi{⟨true stuff⟩}{⟨false stuff⟩}
```

The delimiting ensures that the test will be properly skipped over inside another conditional.

```
\def\@test#1\fi{#1\expandafter\@firstoftwo\else\expandafter\@secondoftwo\fi}
```

Performs a series of `\@test` cases in the form

```
\@testcase
  \if...\fi{Do this}%
  \if...\fi{Do this}%
  \if...\fi{Do this}%
  \default{Do this otherwise}%
```

```
\def\@testcase#1\fi#2#3\default#4{%
    \@test#1\fi{#2}{\ifstrempty{#3}{#4}{\@testcase#3\default{#4}}}%
}
```

2 Expansion Control

Although, there are several useful macros defined in this section, the notable one is `\@expand`, which expands a token multiple times. It takes three arguments: the callback to be executed when done expanding, the stuff to expand, and a count of how many expansions are to be performed, represented by the number of tokens in the argument. Thus:

```
\def\macro{Hello World}
\@expand{\process}{\csname macro\endcsname}{ii}
```

would expand `\csname macro\endcsname` twice (once for each "i"), obtaining the meaning of `\macro`, and then run the callback `\process`, to result in:

```
\process{Hello World}
```

```
\long\def\@expand#1#2#3{\@@expand{#2}{#1}#3\@stop}
\long\def\@@expand#1#2#3#4\@stop{%
    \ifstrempty{#4}{%
\expandafter\@@@expand\expandafter{#1}{#2}%
    }{%
\expandafter\@@expand\expandafter{#1}{#2}#4\@stop
    }%
}
\long\def\@@@expand#1#2{#2{#1}}
```

Returns the first of three groups.

```
\def\@firstofthree#1#2#3{#1}
```

Expands #2 once and provides it as an argument to #1 (which must be a single token macro).

```
\def\@expandarg#1#2{\expandafter#1\expandafter{#2}}%
```

Expands #2 and #3 once, and runs #1 with the expansions as arguments.

```
\def\@expandtwo#1#2#3{%
    \expandafter\@@expandtwo\expandafter{#2}{#3}{#1}%
}
\def\@@expandtwo#1#2#3{%
    \expandafter\@@@expandtwo\expandafter{#2}{#1}{#3}%
}
\def\@@@expandtwo#1#2#3{%
    #3{#2}{#1}%
}
```

These macros are tokens used to delimit parts of macro arguments.

```
\let\str@mark\relax
\let\str@stop\relax
\def\str@test{\str@stop}
```

Removes braces from the arguments. This is useful if you have a callback that takes a series of arguments, and you use one of the functions like `\@expand` that puts braces around the callback result.

```
\def\@unbrace#1#2{#1#2}
```

A useful helper for `\find@in`: takes two macro names and two definitions, and assigns the macro names to the definitions.

```
\def\@twodef#1#2#3#4{%
    \def#1{#3}\def#2{#4}%
}
```

Takes two callbacks and two parameters, and runs each callback on the corresponding parameter.

```
\def\@tworun#1#2#3#4{%
    #1{#3}#2{#4}%
}
```

Pads a string #1 with spaces so that its length equals the length of string #2. Fully expandable.

```
\def\pad@string#1#2{%
    \pad@string@#1\str@mark#2\str@stop{}%
}
\def\pad@string@#1#2\str@mark#3#4\str@stop#5{%
    \ifstrempty{#4}{%
        #5#1#2%
    }{%
        \ifstrempty{#2}{%
            \pad@string@{ }%
        }{%
            \pad@string@#2%
        }\str@mark#4\str@stop{#5#1}%
    }%
}
```

3 Adding to Macros

These macros could all be replaced with `etoolbox` command equivalents.

Adds to a macro.

```
\def\addto@macro#1#2{\expandafter\def\expandafter#1\expandafter{#1#2}}
```

Adds to a macro, but instantiates the macro as empty if it didn't exist before.

```
\def\addto@list#1#2{%
    \@ifundefined{\expandafter\@gobble\string#1}{%
        \def#1{#2}%
    }{\addto@macro#1{#2}}%
}
```

Prepends to a macro.

```
\def\prepend@macro#1#2{%
    \expandafter\prepend@macro@\expandafter{#1}{#1}{#2}%
}
\long\def\prepend@macro@#1#2#3{\def#2{#3#1}}
```

Adds the contents of a macro to a macro.

```
\def\add@macro@to@macro#1#2{%
    \expandafter\addto@macro\expandafter#1\expandafter{#2}%
}
```

If the list is undefined or empty, adds the given element. Otherwise, adds a space and the element.

```
\def\spcappto#1#2{%
    \ifx#1\relax \def#1{#2}\else
        \ifx#1\@empty \def#1{#2}\else
            \appto#1{ #2}%
        \fi
    \fi
}
```

4 Finders

These macros provide convenient ways of defining text finders that, when called, are fully expandable. They provide the ability to find text ⟨*needle*⟩ at the beginning, middle, or end of text ⟨*haystack*⟩. The pattern is to call:

`\make@find@`⟨*type*⟩`{`⟨*needle*⟩`}`

to initialize the finder, and then:

`\find@`⟨*type*⟩`{`⟨*needle*⟩`}{`⟨*haystack*⟩`}`
`{`⟨*true-callback*⟩`}{`⟨*false-callback*⟩`}`

to execute the finder. ⟨*type*⟩ is `in`, `start`, `end`, or `eq` depending on the type of matching desired.

The callbacks for finders are given consistent arguments. The ⟨*false-callback*⟩ always receives no arguments. The ⟨*true-callback*⟩, however, receives arguments indicating the remaining unmatched text. Thus, for `\find@in`, the ⟨*true-callback*⟩ receives two arguments, the prefix and suffix relative to the matched text. However, the callbacks for `\find@start` and `\find@end` receive only one argument, and `\find@eq`'s callback receives none.

The design of finders is as follows. To make a finder, a macro is defined with an argument template to match the text to be found. That macro is called with the text to be searched (with some additional trimmings to help determine whether the finder macro matched).

This is a helper command that other finders use, which invokes the internal finding macro.

```
\def\find@use#1#2#3{%
    \@ifundefined{finder@#1@#2}{%
        \PackageError{strings}{Finder of type #1 for `#2' not defined; used}{%
            You must run \@backslashchar make@find@#1{#2} first%
        }\@gobbletwo
    }{%
        \@nameuse{finder@#1@#2}#3%
    }%
}
```

Makes a finder that searches for text anywhere within a string.

The actual finder macro takes input of the form:

```
\finder@in@⟨text⟩
    ⟨haystack⟩\str@mark
    ⟨needle⟩\str@mark\str@stop
    {⟨text-to-search⟩}
    {⟨callback-if-found⟩}
    {⟨callback-if-not-found⟩}
```

```
\def\make@find@in#1{%
    \@ifundefined{finder@in@#1}{%
        \global\@namedef{finder@in@#1}##1#1##2\str@mark##3\str@stop##4##5{%
            \ifstrempty{##3}{##5}{##4{##1}{##2}}%
        }%
    }{}%
}
```

Finds text using a predefined finder. Takes four arguments: the text to find, the text to be searched, the callback if #1 is found, and the callback if #1 is not found. The first callback takes two arguments, which are the pre-match text and post-match text; the latter callback takes no arguments.

```
\def\find@in#1#2{%
    \find@use{in}{#1}{#2\str@mark#1\str@mark\str@stop}%
}
```

Like `\find@in`, but finds the last instance of #1 in #2 rather than the first. Because this has to iterate against all of #2, this will likely mess up internal grouping in #2.

```
\def\find@last#1#2{%
    \find@in{#1}{#2}{%
        \find@last@{#1}%
    }{\@secondoftwo}%
}
\def\find@last@#1#2#3{%
    \find@in{#1}{#3}{\find@last@@{#1}{#2}}{\find@last@@@{#2}{#3}}%
}
\def\find@last@@#1#2#3#4{%
    \find@in{#1}{#4}{\find@last@@{#1}{#2#1#3}}{\find@last@@@{#2#1#3}{#4}}%
}
\def\find@last@@@#1#2#3#4{#3{#1}{#2}}
```

Makes a named finder with name #1, so #2 can contain control sequence names.

```
\def\make@find@in@cs#1#2{%
    %
    % The actual finder macro takes input of the form:
    % \finder@in@[text]
    %     [text-to-search]\str@mark
    %     [text-to-find]\str@mark\str@stop
    %     {[text-to-search]}{[callback-if-found]}{[callback-if-not-found]}
    \@ifundefined{finder@cs@in@#1}{%
        \global\@namedef{finder@cs@in@#1}##1#2##2\str@mark##3\str@stop##4##5{%
            \ifstrempty{##3}{##5}{##4{##1}{##2}}%
        }%
        \global\@namedef{findcs@in@#1}##1{%
            \@nameuse{finder@cs@in@#1}##1\str@mark#2\str@mark\str@stop
        }%
    }{}%
}
```

The equivalent of `\find@in` except for command sequences.

```
\def\find@in@cs#1#2{%
    \@nameuse{findcs@in@#1}{#2}%
}
```

Makes finders that require the two strings to be equal.

```
\def\make@find@eq#1{%
    \@ifundefined{finder@eq@#1}{%
        %
        % This finder expects to be called as:
        % \finder@eq@[text]
        %     [text-to-compare]\str@mark
        %     [text-of-finder]\str@mark\str@stop
        %     {macro-if-yes}{macro-if-no}
        % Both macros take no arguments.
        \global\@namedef{finder@eq@#1}##1#1##2\str@mark##3\str@stop##4##5{%
            \ifstrempty{##1##2}{##4}{##5}%
        }%
    }{}%
}
```

Tests if the two strings are equal. Takes four arguments: the two strings to compare, a macro if they are equal, and a macro if they are not (the two macros taking no arguments).

```
\def\find@eq#1#2{%
    \find@use{eq}{#1}{#2\str@mark#1\str@mark\str@stop}%
}
```

Makes a finder that finds a text at the end of a string.

```
\def\make@find@end#1{%
    \@ifundefined{finder@end@#1}{%
        \global\@namedef{finder@end@#1}##1#1\str@mark##2\str@stop##3##4{%
            \ifstrempty{##2}{##4}{##3{##1}}%
        }%
    }{}%
}
```

Finds text at the end of a string. Takes four arguments, the first two being the string to find and the string to be searched. Upon a match, execute #3 with one argument, the front matter of the string. Otherwise, execute #4 with no argument.

```
\def\find@end#1#2{%
    \find@use{end}{#1}{#2\str@mark#1\str@mark\str@stop}%
}
```

Makes a finder that finds a text at the start of a string.

```
\def\make@find@start#1{%
    \@ifundefined{finder@start@#1}{%
        \global\@namedef{%
            finder@start@#1%
        }##1\str@mark#1##2\str@mark##3\str@stop##4##5{%
            \ifstrempty{##1}{##4{##2}}{##5}%
        }%
    }{}%
}
```

Finds text at the start of a string. Takes four arguments, the first two being the string to find and the string to be searched. Upon a match, execute #3 with one argument, the trailing matter of the string. Otherwise, execute #4 with no argument.

```
\def\find@start#1#2{%
    \find@use{start}{#1}{\str@mark#2\str@mark#1\str@mark\str@stop}%
}
```

Defines a command sequence start finder. #1 is a nickname for the finder, #2 the sequence to search for.

```
\def\make@find@start@cs#1#2{%
    \@ifundefined{finder@cs@start@#1}{%
        \global\@namedef{%
            finder@cs@start@#1%
        }##1\str@mark#2##2\str@mark##3\str@stop##4##5{%
            \ifstrempty{##1}{##4{##2}}{##5}%
        }%
        \global\@namedef{findcs@start@#1}##1{%
            \@nameuse{finder@cs@start@#1}%
                \str@mark##1\str@mark#2\str@mark\str@stop
        }%
    }{}%
}
\def\find@start@cs#1#2{\@nameuse{findcs@start@#1}{#2}}
```

Iterates over a series of possible finders, running a function upon success. The arguments are:

#1 is the find command (`\find@in`, `\find@start` etc.)

#2 is a list of pairs of parameters as explained below.

#3 is the text to be searched

#4 is a callback if none of the matches succeed. It should take no arguments.

The parameter pairs argument looks as follows:

```
{%
    {⟨needle-1⟩}{⟨true-callback-1⟩}
    {⟨needle-2⟩}{⟨true-callback-2⟩}
    {⟨needle-3⟩}{⟨true-callback-3⟩}
    …
}
```

The callbacks receive the number of arguments corresponding to the finder command (#1).

```
\def\find@try#1#2#3#4{%
    \find@try@{#1}#2\str@stop{#3}{#4}%
}
% This is a table mapping arguments to this \find@try@ helper.
% \find@try  \find@try@
% #1         #1
% #2, pair   #2, #3
% #2, rest   #4
% #3         #5
% #4         #6
\def\find@try@#1#2#3#4\str@stop#5#6{%
    #1{#2}{#5}{#3}{%
        \ifstrempty{#4}{#6}{\find@try@{#1}#4\str@stop{#5}{#6}}%
    }%
}
```

Chops space off of a string. Fully expandable.

```
\newcommand*{\chop@space@then@run}[2]{%
    \find@start{ }{#1}{\chop@space@then@run}{\str@chopspc@end{#1}}{#2}%
}
\def\str@chopspc@end#1#2{%
    \find@end{ }{#1}{\str@chopspc@end}{\str@chopspc@run{#1}}{#2}%
}
\def\str@chopspc@run#1#2{#2{#1}}
\newcommand*{\chop@space}[1]{\chop@space@then@run{#1}{\@iden}}
\make@find@start{ }
\make@find@end{ }
```

5 Cardinal Numbers

The `\cardinaltext` macro changes a number into its capitalized, cardinal textual form. For example, “123” becomes “One Hundred Twenty-Three”.

```
\def\cardinaltext#1{%
    \ifnum#1<100
        \cardinaltext@digits#1\@stop
    \else
        #1
    \fi
}
\def\cardinaltext@digits#1#2\@stop{%
    \ifstrempty{#2}{%
        \cardinaltext@ones{#1}%
    }{%
        \ifnum#1=\@ne
            \cardinaltext@teens{#2}%
        \else
            \cardinaltext@tens{#1}%
            \ifnum#2>\z@
                -\cardinaltext@ones{#2}%
            \fi
        \fi
    }%
}
\def\cardinaltext@ones#1{%
    \ifcase#1Zero\or One\or Two\or Three\or Four\or Five\or Six\or Seven\or
    Eight\or Nine\fi
}
\def\cardinaltext@teens#1{%
    \ifcase#1Ten\or Eleven\or Twelve\or Thirteen\or Fourteen\or Fifteen\or
    Sixteen\or Seventeen\or Eighteen\or Nineteen\fi
}
\def\cardinaltext@tens#1{%
    \ifcase#1\or\or Twenty\or Thirty\or Forty\or Fifty\or Sixty\or Seventy\or
    Eighty\or Ninety\fi
}
```

6 Text Tokenizer

Often it is useful to tokenize a string of TEX tokens into words, in a way that properly handles macros and groups, and differentiates the tokens by type. The following macros help to do that.

The `\find@next` macro finds the next single token by type and runs a command on it. This macro can detect three types of things at the beginning of a string:

- A space
- A group
- A character

For each detected item, a callback macro is run with two arguments, the found item and the remainder of the string (except for the callback when nothing is left; this receives no arguments).

#1 is the string to test, #2 is the callback for spaces, #3 for groups, #4 for characters, #5 for when there is nothing left.

XXX: These functions do not appear to handle groups at the ends of strings correctly.

```
\def\find@next#1#2#3#4#5{%
    \find@start{ }{#1}{#2{ }}{%
        \ifstrempty{#1}{#5}{%
```

```
            \find@next@group#1\str@mark{}\str@stop{#3}{#4}%
        }%
    }%
}
\def\find@next@group#1#{%
    \ifstrempty{#1}\find@next@group@yes{\find@next@group@no#1}%
}
\def\find@next@group@yes#1#2\str@mark#3\str@stop#4#5{#4{#1}{#2}}
\def\find@next@group@no#1#2\str@mark#3\str@stop#4#5{%
    #5{#1}{#2}%
}
```

Finds the next word in the given string and runs a callback on it. This is like `\find@next` except (1) it aggregates letters into a single word, and (2) it differentiates letters from other characters.

#1 is the string to test, #2 is the callback for non-letter characters (including spaces), #3 for groups, #4 for words, #5 for when there is nothing left.

```
\def\find@word#1#2#3#4#5{%
    \find@next{#1}{#2}{#3}{\find@word@char{#2}{#4}}{#5}%
}
```

This is called if the first character of the string is a character. Tests if it's a letter. If so, then runs `\find@word@next` on the next character, specifying that (1) if the subsequent character is also a letter then run `\find@word@acc` to continue accumulating a word, but (2) if the subsequent character is not a letter then run the word callback.

#1 is the punctuation callback; #2 is the word callback; #3 is the character most recently read; #4 is the remainder of the text.

```
\def\find@word@char#1#2#3#4{%
    \@ifwordchar{#3}{%
        \find@word@next{#4}{#2{#3}{#4}}{\find@word@acc{#2}{#3}{#4}}%
    }{%
        #1{#3}{#4}% First char of the string was punctuation
    }%
}
```

Runs `\find@next` on #1, calling #2 with no args on anything other than a character and #3 with args on a character.

```
\def\find@word@next#1#2#3{%
    \find@next{#1}%
        {\@firstofthree{#2}}{\@firstofthree{#2}}{#3}{#2}%
}
```

Knowing that at least one word character has been found, continues reading the text to find all the word characters, calling itself recursively to do so. #1 is the word callback; #2 is the accumulated word so far; #3 is the full remaining text after #2; #4 is the first character of #3, and #5 is the text after the first character of #3.

```
\def\find@word@acc#1#2#3#4#5{%
    \@ifwordchar{#4}{%
        \find@word@next{#5}{#1{#2#4}{#5}}{\find@word@acc{#1}{#2#4}{#5}}%
    }{%
        #1{#2}{#3}%
    }%
}
```

7 Searching for Braced Groups

The `\find@group` macro finds a braced group in a string and runs a callback with three arguments: the pre-group text, the group (without braces), and the post-group text.

The `\find@group@end` macro similarly finds a braced group, but only at the end of a string. The callback receives only two arguments.

#1 is the text, #2 the callback, and #3 a callback if no braced group is found.

```
\def\find@group#1#2#3{%
    \find@group@#1\str@mark{.}\str@stop{#2}{#3}%
}
\def\find@group@#1#{\find@group@@{#1}}
```

#1 is the pre-group text, #2 the group, #3 the post-group text (possibly with `\str@mark`), #4 and #5 callbacks.

```
\def\find@group@@#1#2#3\str@stop#4#5{%
    \ifstrempty{#3}{#5}{\find@group@yes{#1}{#2}#3\str@stop{#4}}%
}
```

#1-#3 are the pre-, in-, and post-group text, #4 the empty material, #5 the callback.

```
\def\find@group@yes#1#2#3\str@mark#4\str@stop#5{#5{#1}{#2}{#3}}
```

Finds a braced group at the end of a text. Runs a callback with two arguments, the pre-group text and the group. #1 is the text, #2 the callback with a group, and #3 the callback if no group.

```
\def\find@group@end#1#2#3{%
    \find@group@end@{}{#1}{#2}{#3}%
}
```

#1 is the already-processed text, #2 the text remaining to process, #3 and #4 the callbacks.

```
\def\find@group@end@#1#2#3#4{%
    \find@group{#2}{%
        \find@group@end@@{#1}{#3}{#4}%
    }{#4}%
}
```

#1 is the already-processed text, #2-3 are the callbacks; #4-#6 the pre-, in-, and post-group text.

```
\def\find@group@end@@#1#2#3#4#5#6{%
    \ifstrempty{#6}{#2{#1#4}{#5}}{%
        \find@group@end@{#1#4{#5}}{#6}{#2}{#3}%
    }%
}
```

8 Testing Text Types

Tests if something is a character.

```
\def\@ifletter#1{\@test \ifcat a\noexpand#1\fi}
```

Tests whether a string is a single character. If it is zero characters, this function treats it as false. A macro will count as a single character.

```
\def\@ifonechar#1{\ifstrempty{#1}{\@secondoftwo}{\@ifonechar@#1\str@stop}}
\def\@ifonechar@#1#2\str@stop{\ifstrempty{#2}}
```

Tests if something is a single digit token.

```
\def\@ifdigit#1{\@ifonechar{#1}{\@test\ifnum 9<1\noexpand#1 \fi}{\@secondoftwo}}
```

Tests if something is a word character (letter or number).

```
\def\@ifwordchar#1{%
    \@ifletter{#1}\@firstoftwo{\@ifdigit{#1}\@firstoftwo\@secondoftwo}%
}
```

Tests if something is a number. A negative number is permitted.

```
\def\@ifnumber#1{%
    \ifblank{#1}\@secondoftwo{%
        \find@start{-}{#1}{\@unbrace\@ifnumber@}{%
            \@ifnumber@#1%
        }\@stop
    }%
}
\make@find@start{-}
\def\@ifnumber@#1#2\@stop{%
    \@ifdigit{#1}{%
        \ifblank{#2}\@firstoftwo{\@ifnumber@#2\@stop}%
    }\@secondoftwo
}
```

Tests whether a string ends with a letter.

```
\def\@ifendswithwordchar#1{%
    \ifstrempty{#1}{\@secondoftwo}{%
        \find@end{ }{#1}{\@thirdofthree}{%
            \str@ifewwc#1\str@stop
        }%
    }%
}
\def\str@ifewwc#1#2\str@stop{%
    \ifstrempty{#2}{%
        \@ifonechar{#1}{%
            \@ifwordchar{#1}%
        }{%
            \str@ifewwc#1\str@stop
        }%
    }{%
        \str@ifewwc#2\str@stop
    }%
}
```

Tests whether a string starts with a letter.

```
\def\@ifstartswithwordchar#1{%
    \ifstrempty{#1}{\@secondoftwo}{%
        \find@start{ }{#1}{\@thirdofthree}{%
            \@ifonechar{#1}{%
                \@ifwordchar{#1}%
            }{%
                \expandafter\@ifstartswithwordchar\expandafter{\@car#1\@nil}%
            }%
        }%
    }%
}
```