The stringstrings Package

Extensive array of string manipulation routines for cosmetic and programming application

Steven B. Segletes
steven@arl.army.mil

2008/03/28
v1.00

Abstract

The stringstrings package provides a large and sundry array of routines for the manipulation of strings. The routines are developed not only for cosmetic application, such as the changing of letter cases, selective removal of character classes, and string substitution, but also for programming application, such as character look-ahead applications, argument parsing, `\if`-tests for various string conditions, etc. A key tenet employed during the development of this package (unlike, for comparison, the `\uppercase` and `\lowercase` routines) was to have resultant strings be "expanded" (*i.e.*, the product of an `\edef`), so that the stringstrings routines could be strung together sequentially and nested (after a fashion) to achieve very complex manipulations.

Contents

1 Motivation

There were two seminal moments that brought about my motivation to develop this package. The first was the realization of the oft cited and infamous LaTeX limitation concerning the inability to nest letter-case changes with LaTeX's intrinsic `\uppercase` and `\lowercase` routines. The second, though not diminishing its utility in many useful applications, was the inherent limitations of the coolstr package, which is otherwise a useful tool for extracting substrings and measuring string lengths.

The former is well documented and need not be delved into in great detail. Basically, as it was explained to me, `\uppercase` and `\lowercase` are expanded by LaTeX at the last possible moment, and thus attempts to capture their result for subsequent use are doomed to failure. One is forced to adopt the left-to-right (rather than nested) approach to case changes.

In the case of the coolstr package, I again want to express my admiration for the utility of this package. I briefly considered building the stringstrings package around it, but it proved unworkable, because of some intrinsic limitations. First, coolstr operates on strings, not tokens, and so in order to fool it into working on tokenized inputs, one must use the cumbersome nomenclature of

```
\expandafter\substr\expandafter{\TokenizedString}{...}{...}
```

in order to, for example grab a substring of `\TokenizedString`. One may `\def` the result of this subroutine, and use it elsewhere in an unaltered state. However, one may not expand, via `\edef`, the result of `\substr` in order to use it as input to a subsequent string manipulation. And thus, the desire to engage in successive string manipulations of different natures (*e.g.*, capitalization of leading characters, extraction of words, reversal of character sequence, removal of character classes, etc., etc.) are not achievable in the context of coolstr.

It was this state of affairs that brought me to hunger for routines that could thoroughly manipulate strings, and yet produce their result "in the clear" (*i.e.*, in an untokenized form) which could be used as input for the next manipulation. It turns out the heart of the stringstrings package which achieves this goal is based on the simple (if much maligned) `\if` construct of LaTeX, by using successive iterations of the following construct:

```
\if [test char.][string][manipulated test char.]\else ...\fi
```

in which a character at the begining of a string is tested. If a match is found, the manipulated test character is replaced at the end of the string, while the original test character is lopped off from the beginning of the string. A false result is used to proceed to a different test character. In this manner, the string may be rotated through, character by character, performing the desired manipulations. And, most importantly, the product of this operation may be placed into an `\edef`.

It turns out there was one glitch to this process (which has been successfully remedied in the stringstrings package). And that is that there are several tokenized LaTeX symbols (*e.g.*, `\$`, `\{`, `\}`, `\AE`, `\oe`, etc.) which expand to more than a single character. If I was more savvy on LaTeX constructs, I would probably have known how to handle this better. But my solution was to develop my own encoding scheme wherein these problematic characters were re-encoded in my intermediate calculations as a 2-byte (escape character-escape code) combination, and only converted back into LaTeX symbols at the last moment, as the finalized strings were handed back to the user.

There are also several tokens, like `\dag`, `\ddag`, and `\P` which can not be put into an `\edef` construct. The solution developed for strings containing these such characters was to convert the encoded string not into an expanded `\edef` construct, but rather back into a tokenized form amenable to `\def`. The `\retokenize` command accomplishes this task and several others.

There was also one glitch that I have not yet been able to resolve to my full satisfaction, though I have provided a workaround. And that is the occurance of LaTeX grouping characters, `{` and `}`, that might typically occur in math mode. The problem is that the character-rotate technique that is the core of stringstrings breaks when rotating these group characters. Why?? Because a string comprised of `...{...}...`, during the rotation process, will eventually become `...}.....{` during an intermediate stage of character rotation. This latter string breaks LaTeX because it is not a properly constructed grouping, even if subsequent rotations would intend to bring it back into a proper construction.

And so, while stringstrings can handle certain math-mode constructs (*e.g.*, `$`, `^`, and `_`), it is unable to *directly* handle groupings that are brought about by the use of curly braces. Note that `\{` and `\}` are handled just fine, but not `{` and `}`. As a result of this limitation regarding the use of grouping braces within strings, stringstrings support for various math symbols remains quite limited.

While it is also common to use curly braces to delimit the arguments of diacritical marks in words like `m\"{u}de` *etc.*, the same result can be achieved without the use of braces as `m\"ude`, with the proper result obtained: müde. For diacritical marks that have an alphabetic token such as the breve, given by `\u`, the curly braces can also be omitted, with the only change being a space required after the `\u` to delimit the token. Thus, `c\u at` becomes căt. Therefore, when manipulating strings containing diacritical marks, it is best to formulate them, if possible, without the use of curly braces.

The workaround fix I have developed, to provide grouping functions within stringstrings arguments, involves the use of newly defined tokens `\LB` and `\RB` (to be used in lieu of `{` and `}`), along with a command `\retokenize`. This workaround will be described subsequently, in the Disclaimers section.

2 Philosophy of Operation

There are several classes of commands that have been developed as part of the stringstrings package. In addition to **Configuration Commands**, which set parameters for subsequent string operations, there are the following commands classes:

- **Commands to Manipulate Strings** – these commands take an input string or token and perform a specific manipulation on the string;
- **Commands to Extract String Information** – these commands take an input string or token, and ascertain a particular characteristic of the string; and
- **Commands to Test Strings** – these commands take an input string or token and test for a particular alphanumeric condition.

Of course, there are also **Support Commands** which are low-level routines which provide functionality to the package, which are generally not accessible to the user.

To support the intended philosophy that the user may achieve a complex string manipulation though a series of simpler manipulations (which is otherwise known as nesting), a mechanism had to be developed. True command nesting of the form `\commandA{\commandB{\commandC{string}}}` is **not** supported by the stringstrings package, since many of the manipulation commands make use of (and would thus inadvertantly overwrite) the same sets of variables used by other routines. Furthermore, there is that 'ol left-to-right philosophy of LaTeX to contend with.

`\thestring`

Instead, in the case of commands that manipulate strings, the expanded (*i.e.*, `edef`'ed) result of the manipulation is placed into a string called `\thestring`. Then, `\thestring` may either be directly used as the input to a subsequent operation, or `edef`'ed into another variable to save it for future use.

`[v]` `[q]`

String manipulation commands use an optional first argument to specify what to do with the manipulated string (in addition to putting it in `\thestring`). Most string manipulation commands default to verbose mode `[v]`, and print out their result immediately on the assumption that a simple string manipulation is, many times, all that is required. If the user wishes to use the manipulated result as is, but needs to use it later in the document, a quiet mode `[q]` is provided which suppresses the immediate output of `\thestring`.

In the absence of symbol tokens (*e.g.*, `\$`, `\&`, `\oe`, `\^`, *etc.*), the verbose and quiet modes would prove sufficient. However, when a tokenized symbol is `\edef`'ed, the token is expanded to the actual symbolic representation of the character. If this expanded symbol is used as part of an input string to a subsequent stringstrings manipulation routine, it gets confused, because the means to detect the token are characteristically different than the means to detect the expanded

symbol. Thus, if one wishes to use `\thestring` as an input to a subsequent manipulation routine, stringstrings provides an encoded mode `[e]` which places an encoded version of the resulting manipulation into `\thestring`. The encoded mode is also a quiet mode, since it leaves `\thestring` in a visually unappealing state that is intended for subsequent manipulation.

`[e]`

The encoded mode is not a LaTeX standard, but was developed for this application. And therefore, if the result of a stringstrings manipulation is needed as input for a routine outside of the stringstrings package, the encoded mode will be of no use. For this reason (and others), the `\retokenize` command is provided. Its use is one of only three times that a stringstrings command returns a tokenized `\def`'ed string in `\thestring`, rather than an expanded, `edef`'ed string. And in the other two cases, both call upon `\retokenize`.

`\retokenize`

In addition to providing tokenized strings that can be passed to other programs, `\retokenize` can also remedy stringstrings problems associated with inadequate character encodings (OT1) and the use of grouping characters `{` and `}` within stringstrings arguments. This issue is discussed more fully in the Disclaimers section, and in the actual `\retokenize` command description.

Therefore, for complex multistage string manipulations, the recommended procedure is to perform each stage of the manipulation in encoded `[e]` mode, passing along `\thestring` to each subsequent stage of the manipulation, until the very last manipulation, which should be, at the last, performed in verbose `[v]` or quiet `[q]` modes. If the resulting manipulation is to be passed to a command outside of the stringstrings package for further manipulation (or if the string contains characters which cannot be placed into an `\edef`), `\thestring` may need to be `\retokenize`'ed. If concatenations of two (or more) different manipulations are to be used as input to a third manipulation, `\thestring` from the first manipulation will need to be immediately `\edef`'ed into a different variable, since `\thestring` will be overwritten by the second manipulation.

Moving on to commands that extract string information, this class of commands (unless otherwise noted) output their result into a token named `\theresult`. It is not a manipulated form of the string, but rather a piece of information about the string, such as "how many characters are in the string?", "how many words are in the string?", "how many letter 'e's are in the string?", *etc.*

The final class of stringstrings commands are the string-test commands. While some of this class of commands also store their test result in `\theresult`, many of these commands use the `\test`*condition*`{`*string*`}` `\if`*condition* constructs (see ifthen package) to answer true/false questions like "is the string composed entirely of lowercase characters?", "is the string's first letter capitalized?" *etc.*

3 Configuration Commands

\Treatments{*U-mode*}{*l-mode*}{*p-mode*}{*n-mode*}{*s-mode*}{*b-mode*}
\defaultTreatments
\+
\?

\Treatments The command \Treatments is used to define how different classes of characters are to be treated by the command \substring, which is the brains of the stringstrings package. As will be explained in the next section, most string manipulation routines end up calling \substring, the difference between them being a matter of how these character treatments are set prior to the call. Because most string manipulation commands will set the treatments as necessary to perform their given task, and reset them to the default upon conclusion, one should set the \Treatments immediately prior to the call upon \substring.

\Treatments has six arguments, that define the mode of treatment for the six classes of characters that stringstrings has designated. All modes are one-digit integers. They are described below:

- *U-mode*— This mode defines the treatment for the upper-case characters (A–Z, Œ, Æ, and Å). A mode of 0 tells \substring to remove upper-case characters, a mode of 1 indicates to leave upper-case characters alone, and a mode of 2 indicates to change the case of upper-case characters to lower case.

- *l-mode*— This mode defines the treatment for the lower-case characters (a–z, œ, æ, å, and ß). A mode of 0 tells \substring to remove lower-case characters, a mode of 1 indicates to leave lower-case characters alone, and a mode of 2 indicates to change the case of lower-case characters to upper case. In the case of the eszett character (ß), there is no uppercase equivalent, and so an *l-mode* of 2 will leave the eszett unchanged.

- *p-mode*— This mode defines the treatment for the punctuation characters. stringstrings defines the punctuation characters as ; : ’ ” , . ? ‘ and ! A mode of 0 tells \substring to remove punctuation characters, while a mode of 1 indicates to leave punctuation characters as is.

- *n-mode*— This mode defines the treatment for the numerals (0–9). A mode of 0 tells \substring to remove numerals, while a mode of 1 indicates to leave numerals as is.

- *s-mode*— This mode defines the treatment for the symbols. stringstrings defines symbols as the following characters: / * ( ) - = + [ ] ˇ< > \& \% \# \{ \} _ \$ † ‡ § ¶ as well as \^ @ and the pipe symbol. A mode of 0 tells \substring to remove symbols, while a mode of 1 indicates to leave symbols as is. Note that the $ symbol, when used for entering and exiting math mode, is left intact, regardless of *s-mode*.

- *b-mode*— This mode defines the treatment for blankspaces. A mode of 0 tells `\substring` to remove blankspaces, while a mode of 1 indicates to leave blankspaces as is. The treatment apples to both soft ( ) as well as hard (~) spaces.

`\defaultTreatments` The command `\defaultTreatments` resets all treatment modes to their default settings, which are to leave individual characters unaltered by a string manipulation.

`\+` `\?` The commands `\+` and `\?` are a pair that work in tandem to turn on stringstrings encoding and turn off stringstrings encoding, respectively. Generally, the user will not need these commands unless he is writing his own routines to take advantage of the stringstrings library. After `\+` is called, tokens which would otherwise expand to multi-byte sequences are instead encoded according to the stringstrings methodology. The affected tokens include `\$ \^ \" \{ \} \_ \dag \ddag \P \S \ss \AA \aa \O \o \AE \ae \OE` and `\oe`. In addition, hard spaces (~) are encoded as well. The command `\?` restores the standard LaTeX encoding for these tokens.

4 Commands to Manipulate Strings

These commands take an input string or token and perform a specific manipulation on the string. They include:

`\substring[`*mode*`]{`*string*`}{`*min*`}{`*max*`}`
`\caseupper[`*mode*`]{`*string*`}`
`\caselower[`*mode*`]{`*string*`}`
`\solelyuppercase[`*mode*`]{`*string*`}`
`\solelylowercase[`*mode*`]{`*string*`}`
`\changecase[`*mode*`]{`*string*`}`
`\noblanks[`*mode*`]{`*string*`}`
`\nosymbolsnumerals[`*mode*`]{`*string*`}`
`\alphabetic[`*mode*`]{`*string*`}`
`\capitalize[`*mode*`]{`*string*`}`
`\capitalizewords[`*mode*`]{`*string*`}`
`\reversestring[`*mode*`]{`*string*`}`
`\convertchar[`*mode*`]{`*string*`}{`*from-char*`}{`*to-string*`}`
`\convertword[`*mode*`]{`*string*`}{`*from-string*`}{`*to-string*`}`
`\rotateword[`*mode*`]{`*string*`}`
`\removeword[`*mode*`]{`*string*`}`
`\getnextword[`*mode*`]{`*string*`}`
`\getaword[`*mode*`]{`*string*`}{`*n*`}`
`\rotateleadingspaces[`*mode*`]{`*string*`}`
`\removeleadingspaces[`*mode*`]{`*string*`}`
`\stringencode[`*mode*`]{`*string*`}`
`\stringdecode[`*mode*`]{`*string*`}`

`\gobblechar[`*mode*`]{`*string*`}`
`\gobblechars[`*mode*`]{`*string*`}{`*n*`}`
`\retokenize[`*mode*`]{`*string*`}`

Unless otherwise noted, the *mode* may take one of three values: `[v]` for verbose mode (generally, the default), `[q]` for quiet mode, and `[e]` for encoded mode. In all cases, the result of the operation is stored in `\thestring`. In verbose mode, it is also output immediately (and may be captured by an `\edef`). In quiet mode, no string is output, though the result still resides in `\thestring`. Encoded mode is also a quiet mode. However, the encoded mode saves the string with its stringstrings encodings. Encoded mode indicates that the result is an intermediate result which will be subsequently used as input to another stringstrings manipulation.

`\substring` The command `\substring` is the brains of the stringstrings package, in that most of the commands in this section call upon `\substring` in one form or another. Nominally, the routine returns a substring of *string* between the characters defined by the integers *min* and *max*, inclusive. However, the returned substring is affected by the designated `\Treatments` which have been defined for various classes of characters. Additionally, a shorthand of $ may be used in *min* and *max* to define END-OF-STRING, and the shorthand $–*integer* may be used to define an offset of *integer* relative to the END-OF-STRING.

Regardless of how many bytes a LaTeX token otherwise expands to, or how many characters are in the token name, each LaTeX symbol token counts as a single character for the purposes of defining the substring limits, *min* and *max*.

While the combination of `\Treatments` and `\substring` are sufficient to achieve a wide array of character manipulations, many of those possibilities are useful enough that separate commands have been created to describe them, for convenience. Several of the commands that follow fall into this category.

`\caseupper` The command `\caseupper` takes the input string or token, and converts all lowercase characters in the string to uppercase. All other character classes are left untouched. Default mode is `[v]`.

`\caselower` The command `\caselower` takes the input string or token, and converts all uppercase characters in the string to lowercase. All other character classes are left untouched. Default mode is `[v]`.

`\solelyuppercase` The command `\solelyuppercase` is similar to `\caseupper`, except that all punctuation, numerals, and symbols are discarded from the string. Blankspaces are left alone, and lowercase characters are converted to uppercase. Default mode is `[v]`.

`\solelylowercase` The command `\solelylowercase` is similar to `\caselower`, except that all punctuation, numerals, and symbols are discarded from the string. Blankspaces are left alone, and uppercase characters are converted to lowercase. Default mode is `[v]`.

`\changecase` The command `\changecase` switches lower case to upper case and upper case

to lower case. All other characters are left unchanged. Default mode is `[v]`.

`\noblanks` The command `\noblanks` removes blankspaces (both hard and soft) from a string, while leaving other characters unchanged. Default mode is `[v]`.

`\nosymbolsnumerals` The command `\nosymbolsnumerals` removes symbols and numerals from a string, while leaving other characters unchanged. Default mode is `[v]`.

`\alphabetic` The command `\alphabetic` discards punctuation, symbols, and numerals, while retaining alphabetic characters and blankspaces. Default mode is `[v]`.

`\capitalize` The command `\capitalize` turns the first character of *string* into its upper case, if it is alphabetic. Otherwise, that character will remain unaltered. Default mode is `[v]`.

`\capitalizewords` The command `\capitalizewords` turns the first character of every word in *string* into its upper case, if it is alphabetic. Otherwise, that character will remain unaltered. For the purposes of this command, "the first character of a word" is defined as either the first character of the string, or the first non-blank character that follows one or more blankspaces. Default mode is `[v]`.

`\reversestring` The command `\reversestring` reverses the sequence of characters in a string, such that what started as the first character becomes the last character in the manipulated string, and what started as the last character becomes the first character. Default mode is `[v]`.

`\convertchar` The command `\convertchar` is a substitution command in which a specified match character in the original string (*from-char*) is substituted with a different string (*to-string*). All occurances of *from-char* in the original string are affected. The *from-char* can only be a single character (or tokenized symbol), whereas *to-string* can range from the null-string (*i.e.*, character removal) to a single character (*i.e.*, character substitution) to a complete multi-character string. Default mode is `[v]`.

`\convertword` The command `\convertword` is a substitution command in which a specified match string in the original string (*from-string*) is substituted with a different string (*to-string*). All occurances of *from-string* in the original string are replaced. Default mode is `[v]`.

`\rotateword` The command `\rotateword` takes the first word of *string* (and its leading and trailing spaces) and rotates them to the end of the string. Care must be taken to have a blankspace at the beginning or end of *string* if one wishes to retain a blankspace word separator between the original last word of the string and the original first word which has been rotated to the end of the string. Default mode is `[v]`.

`\removeword` The command `\removeword` removes the first word of *string*, along with any of its leading and trailing spaces. Default mode is `[v]`.

`\getnextword` The command `\getnextword` returns the next word of *string*. In this case, "word" is a sequence of characters delimited either by spaces or by the beginning or end of the string. Default mode is `[v]`.

`\getaword` The command `\getaword` returns a word of *string* defined by the index, *n*. In this case, "word" is a sequence of characters delimited either by spaces or by the first or last characters of the string. If the index, *n*, requested exceeds the number of words available in the string, the index wraps around back to the first argument of the string, such that asking for the tenth word of an eight word string will return the second word of the string. Default mode is `[v]`.

`\rotateleadingspaces` The command `\rotateleadingspaces` takes any leading spaces of the string and rotates them to the end of the string. Default mode is `[v]`.

`\removeleadingspaces` The command `\removeleadingspaces` removes any leading spaces of the string. Default mode is `[v]`.

`\stringencode` The command `\stringencode` returns a copy of the string that has been encoded according to the stringstrings encoding scheme. Because an encoded string is an intermediate result, the default mode for this command is `[e]`.

`\stringdecode` The command `\stringdecode` returns a copy of the string that has been decoded. Default mode is `[v]`.

`\gobblechar` The command `\gobblechar` returns a string in which the first character of *string* has been removed. Unlike the LaTeX system command `\@gobble` which removes the next **byte** in the input stream, `\gobblechar` not only takes an argument as the target of its gobble, but also removes one **character**, regardless of whether that character is a single-byte or multi-byte character. Because this command may have utility outside of the stringstrings environment, the result of this command is retokenized (*i.e.*, `def`'ed) rather than expanded (*i.e.*, `edef`'ed). Default mode is `[q]`. Mode `[e]` is not recognized.

`\gobblechars` The command `\gobblechars` returns a string in which the first *n* characters of *string* have been removed. Like `\gobblechar`, `\gobblechars` removes characters, regardless of whether those characters are single-byte or multi-byte characters. Likewise, the result of this command is retokenized (*i.e.*, `def`'ed) rather than expanded (*i.e.*, `edef`'ed). Default mode is `[q]`. Mode `[e]` is not recognized.

`\retokenize` The command `\retokenize` takes a string that is encoded according to the stringstrings encoding scheme, and repopulates the encoded characters with their LaTeX tokens. This command is particularly useful for exporting a string to a routine outside of the stringstrings library or if the string includes the following characters: `\{`, `\}`, `\|`, `\dag`, `\ddag`, and `\P`. Default mode is `[q]`. Mode `[e]` is not recognized.

5 Commands to Extract String Information

These commands take an input string or token, and ascertain a particular characteristic of the string. They include:

`\stringlength[`*mode*`]{`*string*`}`
`\findchars[`*mode*`]{`*string*`}{`*match-char*`}`

`\findwords[`*mode*`]{`*string*`}{`*match-string*`}`
`\whereischar[`*mode*`]{`*string*`}{`*match-char*`}`
`\whereisword[`*mode*`]{`*string*`}{`*match-string*`}`
`\wordcount[`*mode*`]{`*string*`}`
`\getargs[`*mode*`]{`*string*`}`

Commands in this section return their result in the string `\theresult`, unless otherwise specified. Unless otherwise noted, the *mode* may take one of two values: `[v]` for verbose mode (generally, the default), and `[q]` for quiet mode. In both cases, the result of the operation is stored in `\theresult`. In verbose mode, it is also output immediately (and may be captured by an `\edef`). In quiet mode, no string is output, though the result still resides in `\theresult`.

`\stringlength` The command `\stringlength` returns the length of *string* in characters (not bytes). Default mode is `[v]`.

`\findchars` The command `\findchars` checks to see if the character *match-char* occurs anywhere in *string*. The number of occurances is stored in `\theresult` and, if in verbose mode, printed. If it is desired to find blankspaces, *match-char* should be set to `{~}` and not `{ }`. Default mode is `[v]`.

`\findwords` The command `\findwords` checks to see if the string *match-string* occurs anywhere in *string*. The number of occurances is stored in `\theresult` and, if in verbose mode, printed. If it is desired to find blankspaces, those characters in *match-string* should be set to hardspaces (*i.e.*, tildes) and not softspaces (*i.e.*, blanks), regardless of how they are defined in *string*. Default mode is `[v]`.

`\whereischar` The command `\whereischar` checks to see where the character *match-char* first occurs in *string*. The location of that occurance is stored in `\theresult` and, if in verbose mode, printed. If the character is not found, `\theresult` is set to a value of 0. If it is desired to find blankspaces, *match-char* should be set to `{~}` and not `{ }`. Default mode is `[v]`.

`\whereisword` The command `\whereisword` checks to see where the string *match-string* first occurs in *string*. The location of that occurance is stored in `\theresult` and, if in verbose mode, printed. If *match-string* is not found, `\theresult` is set to a value of 0. If it is desired to find blankspaces, those characters in *match-string* should be set to hardspaces (*i.e.*, tildes) and not softspaces (*i.e.*, blanks), regardless of how they are defined in *string*. Default mode is `[v]`.

`\wordcount` The command `\wordcount` counts the number of space-separated words that occur in *string*. Default mode is `[v]`.

`\getargs` The command `\getargs` mimics the Unix command of the same name, in that it parses *string* to determine how many arguments (*i.e.*, words) are in *string*, and extracts each word into a separate variable. The number of arguments is placed in `\nargs` and the individual arguments are placed in variables of the name `\argi`, `\argii`, `\argiii`, `\argiv`, etc. This command may be used to facilitate simply the use of multiple optional arguments in a LaTeX command, for example `\mycommand[`*option1 option2 option3*`]{`*argument*`}`. In this case, `\mycommand` should exercise `\getargs{#1}`, with the result being that *option1* is stored in

\argi, *etc.* The command \mycommand may then proceed to parse the optional arguments and branch accordingly. Default mode is [q].

6 Commands to Test Strings

These commands take an input string or token and test for a particular alphanumeric condition. They include:

\isnextbyte[*mode*]{*match-byte*}{*string*}
\testmatchingchar{*string*}{*n*}{*match-char*}
\testcapitalized{*string*}
\testuncapitalized{*string*}
\testleadingalpha{*string*}
\testuppercase{*string*}
\testsolelyuppercase{*string*}
\testlowercase{*string*}
\testsolelylowercase{*string*}
\testalphabetic{*string*}

\isnextbyte The command \isnextbyte tests to see if the first byte of *string* equals *match-byte*. It is the only string-testing command in this section which does not use the ifthen test structure for its result. Rather, \isnextbyte returns the result of its test as a T or F in the string \theresult. More importantly, and unlike other stringstrings commands, \isnextbyte is a *byte* test and not a *character* test. This means that, while \isnextbyte operates very efficiently, it cannot be used to directly detect multi-byte characters like \$, \^, \{, \}, _, \dag, \ddag, \AE, \ae, \OE, \oe, *etc.* (\isnextbyte will give false positives or negatives when testing for these multi-byte characters). The default mode of \isnextbyte is [v].

\testmatchingchar If a character needs to be tested, rather than a byte, \testmatchingchar should be used. The command \testmatchingchar is used to ascertain whether character *n* of *string* equals *match-char* or not. Whereas \isnextbyte checks only a *byte*, \testmatchingchar tests for a *character* (single- or multi-byte character). After the test is called, the action(s) may be called out with \ifmatchingchar *true-code* \else *false-code* \fi.

\testcapitalized The command \testcapitalized is used to ascertain whether the first character of *string* is capitalized or not. If the first character is non-alphabetic, the test will return FALSE. After the test is called, the action(s) may be called out with \ifcapitalized *true-code* \else *false-code* \fi.

\testuncapitalized The command \testuncapitalized is used to ascertain whether the first character of *string* is uncapitalized. If the first character is non-alphabetic, the test will return FALSE. After the test is called, the action(s) may be called out with \ifuncapitalized *true-code* \else *false-code* \fi.

\testleadingalpha The command \testleadingalpha is used to ascertain whether the first character of *string* is alphabetic. After the test is called, the action(s) may be called

out with `\ifleadingalpha` *true-code* `\else` *false-code* `\fi`.

`\testuppercase` The command `\testuppercase` is used to ascertain whether all the alphabetic characters in *string* are uppercase or not. The presence of non-alphabetic characters in *string* does not falsify the test, but are merely ignored. However, a string completely void of alphabetic characters will always test FALSE. After the test is called, the action(s) may be called out with `\ifuppercase` *true-code* `\else` *false-code* `\fi`.

`\testsolelyuppercase` The command `\testsolelyuppercase` is used to ascertain whether *all* the characters in *string* are uppercase or not. The presence of non-alphabetic characters in *string* other than blankspaces will automatically falsify the test. Blankspaces are ignored. However, a null string or a string composed solely of blankspaces will also test FALSE. After the test is called, the action(s) may be called out with `\ifsolelyuppercase` *true-code* `\else` *false-code* `\fi`.

`\testlowercase` The command `\testlowercase` is used to ascertain whether all the alphabetic characters in *string* are lowercase or not. The presence of non-alphabetic characters in *string* does not falsify the test, but are merely ignored. However, a string completely void of alphabetic characters will always test FALSE. After the test is called, the action(s) may be called out with `\iflowercase` *true-code* `\else` *false-code* `\fi`.

`\testsolelylowercase` The command `\testsolelylowercase` is used to ascertain whether *all* the characters in *string* are lowercase or not. The presence of non-alphabetic characters in *string* other than blankspaces will automatically falsify the test. Blankspaces are ignored. However, a null string or a string composed solely of blankspaces will also test FALSE. After the test is called, the action(s) may be called out with `\ifsolelylowercase` *true-code* `\else` *false-code* `\fi`.

`\testalphabetic` The command `\testalphabetic` is used to ascertain whether *all* the characters in *string* are alphabetic or not. The presence of non-alphabetic characters in *string* other than blankspaces will automatically falsify the test. Blankspaces are ignored. However, a null string or a string composed solely of blankspaces will also test FALSE. After the test is called, the action(s) may be called out with `\ifalphabetic` *true-code* `\else` *false-code* `\fi`.

7 Disclaimers

Now that we have described the commands available in the stringstrings package, it is appropriate to lay out the quirks and warnings associated with the use of the package.

First, stringstrings is currently set to handle a string no larger than 500 characters. A user could circumvent this, presumably, by editing their copy of the style `\@MAXSTRINGSIZE` package to increase the value of `\@MAXSTRINGSIZE` .

It is important to remember that stringstrings follows the underlying rules of LaTeX. Therefore, a passed string could not contain a raw % as part of it, because

it would, in fact, comment out the remainder of the line. Naturally, the string may freely contain instances of `\%`.

Tokens that take two or more characters to express (*e.g.*, `\#`, `\oe`, `\ddag`, *etc.*) are **counted as a single character** within the string. The rule applies if you wanted to know the length of a string that was populated with such tokens, or wanted to extract a substring from a such a string. Of course, the exception that makes the rule is that `\^a` counts as two characters, because the `a` is really just the operand of the `\^` token, even though the net result looks like a single character (â).

Consistent with LaTeX convention, groups of spaces are treated as a single blankspace, unless encoded with `~` characters. And finally, again consistent with the way LaTeX operates, the space that follows an alphabetic token is not actually a space in the string, but serves as the delimiter to the token. Therefore, `\OE dipus` (Œdipus) has a length of six characters, one for the `\OE` and five for the `dipus`. The intervening space merely closes out the `\OE` token, and does not represent a space in the middle of the string.

In the `\substring` command, no tests are performed to guarantee that the lower limit, *min*, is less than the upper limit, *max*, or that *min* is even positive. However, the upper limit, *max*, is corrected, if set larger than the string length. Also, the use of the '$' symbol to signify the last character of the string and '$–*n*' to denote an offset of *n* characters from the end of the string can be helpful in avoiding the misindexing of strings.

Table 1 shows a variety of characters and tokens, some of which pose a challenge to stringstrings manipulations. In all cases, a solution or workaround is provided. For symbols in the top two categories, the workaround solution includes the use of retokenized strings instead of expanded strings. For symbols in the next two categories, use of T1 encoding or retokenizing provides a satisfactory solution. In the bottom two categories, because of stringstrings encoded `[e]` mode, there is nothing to impede the use of these characters in stringstrings arguments, if encoded `[e]` mode is employed for intermediate calculations. Some of the details of these problematic cases is described below.

Not surprisingly, you are not allowed to extract a substring of a string, if it breaks in the middle of math mode, because a substring with only one `$` in it cannot be `\edef`'ed.

There are a few potential quirks when using LaTeX's native OT1 character encoding, most of which can be circumvented by using the more modern T1 encoding (accessed via `\renewcommand\encodingdefault{T1}` in the document preamble). The quirks arise because there are several characters that, while displayable in LaTeX, are not part of the OT1 character encoding. The characters include `\{`, `\}`, and the `|` symbol (accessed in stringstrings via `\|`). When using stringstrings to manipulate strings containing these characters in the presence of OT1 encoding, they come out looking like –, ″, and —, respectively. However, if the T1 encoding fix is not an option for you, you can also work around this problem by

Table 1: **Problematic Characters/Tokens and stringstrings Solutions**

LaTeX	Symbol/Name	Problem/Solution
`{`	begin group	Cannot use `{` and `}` in stringstrings arguments.
`}`	end group	However, use `\LB...\RB` in lieu of `{...}`;
		manipulate string in `[e]` mode & `\retokenize`
`\dag`	† Dagger	cannot `\edef` these tokens; Thus, `[v]` mode
`\ddag`	‡ Double Dagger	fails with both OT1 and T1 encoding;
`\P`	¶ Pilcrow	manipulate string in `[e]` mode & `\retokenize`
`\d`	ẋ Underdot	
`\t`	x͡x Joining Arch	
`\b`	x̲ Letter Underline	
`\copyright`	© Copyright	
`\_`	_ Underscore	cannot `\edef` with OT1 encoding; either
`\{`	{ Left Curly Brace	`\renewcommand\encodingdefault{T1}`, or
`\}`	} Right Curly Brace	manipulate string in `[e]` mode & `\retokenize`
`\S`	§ Section Symbol	With OT1, `\S`, `\c` and `\pounds` break
`\c`	ȩ Cedilla	stringstrings `[v]` mode.
`\pounds`	£ Pounds	
`|`	stringstrings Pipe Char.	distinct from `|`, the stringstrings encoded-
	| (T1) — (OT1)	escape character
`\$`	$ Dollar	Either cannot `\edef`, or
`\carat`	^ (text mode)	cannot identify uniquely with `\if` construct, or
`\^`	x̂ Circumflex	expanded character is more than one byte.
`\'`	x́ Acute	
`\"`	ẍ Umlaut	*However*,
`\~`	x̃ Tilde	Use these characters freely, stringstrings
`\‘`	x̀ Grave	encoding functions transparently with them.
`\.`	ẋ Overdot	
`\=`	x̄ Macron	`\retokenize` also works
`\u`	x̆ Breve	
`\v`	x̌ Caron	
`\H`	x̋ Double Acute	
`\ss`	ß Eszett	
`\AE \ae`	Æ æ æsc	
`\OE \oe`	Œ œ œthel	
`\AA \aa`	Å å angstrom	
`\O \o`	Ø ø slashed O	
`\L \l`	Ł ł barred L	
`~`	Hardspace	
`$`	begin/end math mode	These characters pose no difficulties;
`^`	math superscript	However, cannot extract substring that
`_`	math subscript	breaks in middle of math mode.
		Other math mode symbols NOT supported.

`\retokenize`'ing the affected string (the `\retokenize` command is provided to convert encoded, expanded strings back into tokenized form, if need be).

Likewise, for both OT1 and T1 encoding, the characters † (`\dag`), ‡ (`\ddag`), ¶ (`\P`), ̣ (`\d`), ̂ (`\t`), ̲ (`\b`), and © (`\copyright`) cannot be in the argument of an `\edef` expression. For manipulated strings including these characters, `\retokenize` is the only option available to retain the integrity of the string.

As discussed thoroughly in the previous section, an "encoded" form of the string manipulation routines is provided to prevent the undesirable circumstance of passing an `\edef`'ed symbol as input to a subsequent manipulation. Likewise, never try to "decode" an already "decoded" string.

When stringstrings doesn't understand a token, it is supposed to replace it with a period. However, some undecipherable characters may inadvertantly be replaced with a space, instead. Of course, neither of these possibilities is any comfort to the user.

As mentioned already, stringstrings cannot handle curly braces that are used for grouping purposes, a circumstance which often arises in math mode. Nonetheless, `\LB` and `\RB` may be used within stringstrings arguments in lieu of grouping braces, *if the final result is to be retokenized*. Thus, `\caselower[e]{$X^\LB Y + Z\RB$}` followed by `\convertchar[e]{\thestring}{x}{(1+x)}`, when finished up with the following command, `\retokenize[v]{\thestring}` yields as its result: $(1 + x)^{y\ +\ z}$.

One might ask, "why not retokenize everything, instead of using the `[v]` mode of the stringstrings routines?" While one *could* do this, the answer is simply that `\retokenize` is a computationally intensive command, and that it is best used, therefore, only when the more efficient methods will not suffice. In many, if not most cases, strings to be manipulated will be solely composed of alphanumeric characters which don't require the use of `\retokenize`, T1 encoding, or even stringstrings encoding.

Despite these several disclaimers and workarounds required when dealing with problematic characters, I hope you find the stringstrings architecture and feel to be straightforward and useful. There is only one thing left, and that is to dissect the code... and so here we go.

stringstrings.sty

8 Code Listing

I'll try to lay out herein the workings of the stringstrings style package.

```
1 ⟨*package⟩
2
3 %%%%% INITIALIZATIONS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

ifthen

This package makes wide use of the ifthen style package.

```
4 \usepackage{ifthen}
```

\@MAXSTRINGSIZE The parameter \@MAXSTRINGSIZE defines the maximum allowable string size that stringstrings can operate upon.

```
\def\@MAXSTRINGSIZE{500}
```

```
\def\endofstring{@E@o@S@}%
\def\undecipherable{.}% UNDECIPHERABLE TOKENS TO BE REPLACED BY PERIOD
\def\@blankaction{\BlankSpace}
```

Save the symbols which will get redefined stringstrings encoding.

```
\let\SaveDollar\$
\let\SaveHardspace~
\let\SaveCircumflex\^
\let\SaveTilde\~
\let\SaveUmlaut\"
\let\SaveGrave\‘
\let\SaveAcute\’
\let\SaveMacron\=
\let\SaveOverdot\.
\let\SaveBreve\u
\let\SaveCaron\v
\let\SaveDoubleAcute\H
\let\SaveCedilla\c
\let\SaveUnderdot\d
\let\SaveArchJoin\t
\let\SaveLineUnder\b
\let\SaveCopyright\copyright
\let\SavePounds\pounds
\let\SaveLeftBrace\{
\let\SaveRightBrace\}
\let\SaveUnderscore\_
\let\SaveDagger\dag
\let\SaveDoubleDagger\ddag
\let\SaveSectionSymbol\S
\let\SavePilcrow\P
\let\SaveAEsc\AE
\let\Saveaesc\ae
\let\SaveOEthel\OE
\let\Saveoethel\oe
\let\SaveAngstrom\AA
\let\Saveangstrom\aa
\let\SaveSlashedO\O
\let\SaveSlashedo\o
\let\SaveBarredL\L
\let\SaveBarredl\l
\let\SaveEszett\ss
\let\SaveLB{
\let\SaveRB}
```

The BlankSpace character is the only character which is reencoded with a

1-byte re-encoding. . . in this case the Œ character.

```
\def\EncodedBlankSpace{\SaveOEthel}
\edef\BlankSpace{ }
```

All other reencoded symbols consist of 2 bytes: an escape character plus a unique code. The escape character is a pipe symbol. the unique code comprises either a single number, letter, or symbol.

```
\def\EscapeChar{|}

% |0 IS AN ENCODED |, ACCESSED VIA \|
\def\PipeCode{0}
\def\EncodedPipe{\EscapeChar\PipeCode}
\def\Pipe{|}
\let\|\EncodedPipe

% |1 IS AN ENCODED \$
\def\DollarCode{1}
\def\EncodedDollar{\EscapeChar\DollarCode}
% THE FOLLOWING IS NEEDED TO KEEP OT1 ENCODING FROM BREAKING;
% IT PROVIDES AN ADEQUATE BUT NOT IDEAL ENVIRONMENT FOR T1 ENCODING
\def\Dollar{\symbol{36}}
% THE FOLLOWING IS BETTER FOR T1 ENCODING, BUT BREAKS OT1 ENCODING
%\def\Dollar{\SaveDollar}

% |2 IS AN ENCODED ^ FOR USE IN TEXT MODE, ACCESSED VIA \carat
\def\CaratCode{2}
\def\EncodedCarat{\EscapeChar\CaratCode}
\def\Carat{\symbol{94}}
\let\carat\EncodedCarat

% |4 IS AN ENCODED \{
\def\LeftBraceCode{4}
\def\EncodedLeftBrace{\EscapeChar\LeftBraceCode}
% THE FOLLOWING IS NEEDED TO KEEP OT1 ENCODING FROM BREAKING;
% IT PROVIDES AN ADEQUATE BUT NOT IDEAL ENVIRONMENT FOR T1 ENCODING
\def\LeftBrace{\symbol{123}}
% THE FOLLOWING IS BETTER FOR T1 ENCODING, BUT BREAKS OT1 ENCODING
%\def\LeftBrace{\SaveLeftBrace}

% |5 IS AN ENCODED \}
\def\RightBraceCode{5}
\def\EncodedRightBrace{\EscapeChar\RightBraceCode}
% THE FOLLOWING IS NEEDED TO KEEP OT1 ENCODING FROM BREAKING;
% IT PROVIDES AN ADEQUATE BUT NOT IDEAL ENVIRONMENT FOR T1 ENCODING
\def\RightBrace{\symbol{125}}
% THE FOLLOWING IS BETTER FOR T1 ENCODING, BUT BREAKS OT1 ENCODING
%\def\RightBrace{\SaveRightBrace}

% |6 IS AN ENCODED \_
```

```
\def\UnderscoreCode{6}
\def\EncodedUnderscore{\EscapeChar\UnderscoreCode}
\def\Underscore{\symbol{95}}
%\def\Underscore{\SaveUnderscore}

% |7 IS AN ENCODED \^
\def\CircumflexCode{7}
\def\EncodedCircumflex{\EscapeChar\CircumflexCode}
\def\Circumflex{\noexpand\SaveCircumflex}

% |8 IS AN ENCODED \~
\def\TildeCode{8}
\def\EncodedTilde{\EscapeChar\TildeCode}
\def\Tilde{\noexpand\SaveTilde}

% |" IS AN ENCODED \"
\def\UmlautCode{"}
\def\EncodedUmlaut{\EscapeChar\UmlautCode}
\def\Umlaut{\noexpand\SaveUmlaut}

% |‘ IS AN ENCODED \‘
\def\GraveCode{‘}
\def\EncodedGrave{\EscapeChar\GraveCode}
\def\Grave{\noexpand\SaveGrave}

% |’ IS AN ENCODED \’
\def\AcuteCode{’}
\def\EncodedAcute{\EscapeChar\AcuteCode}
\def\Acute{\noexpand\SaveAcute}

% |= IS AN ENCODED \=
\def\MacronCode{=}
\def\EncodedMacron{\EscapeChar\MacronCode}
\def\Macron{\noexpand\SaveMacron}

% |. IS AN ENCODED \.
\def\OverdotCode{.}
\def\EncodedOverdot{\EscapeChar\OverdotCode}
\def\Overdot{\noexpand\SaveOverdot}

% |u IS AN ENCODED \u
\def\BreveCode{u}
\def\EncodedBreve{\EscapeChar\BreveCode}
\def\Breve{\noexpand\SaveBreve}

% |v IS AN ENCODED \v
\def\CaronCode{v}
\def\EncodedCaron{\EscapeChar\CaronCode}
\def\Caron{\noexpand\SaveCaron}

```

```
% |H IS AN ENCODED \H
\def\DoubleAcuteCode{H}
\def\EncodedDoubleAcute{\EscapeChar\DoubleAcuteCode}
\def\DoubleAcute{\noexpand\SaveDoubleAcute}

% |c IS AN ENCODED \c
\def\CedillaCode{c}
\def\EncodedCedilla{\EscapeChar\CedillaCode}
\def\Cedilla{\noexpand\SaveCedilla}

% |d IS AN ENCODED \d
\def\UnderdotCode{d}
\def\EncodedUnderdot{\EscapeChar\UnderdotCode}
\def\Underdot{.}% CANNOT \edef \d

% |t IS AN ENCODED \t
\def\ArchJoinCode{t}
\def\EncodedArchJoin{\EscapeChar\ArchJoinCode}
\def\ArchJoin{.}% CANNOT \edef \t

% |b IS AN ENCODED \b
\def\LineUnderCode{b}
\def\EncodedLineUnder{\EscapeChar\LineUnderCode}
\def\LineUnder{.}% CANNOT \edef \b

% |C IS AN ENCODED \copyright
\def\CopyrightCode{C}
\def\EncodedCopyright{\EscapeChar\CopyrightCode}
\def\Copyright{.}% CANNOT \edef \copyright

% |p IS AN ENCODED \pounds
\def\PoundsCode{p}
\def\EncodedPounds{\EscapeChar\PoundsCode}
\def\Pounds{\SavePounds}

% |[ IS AN ENCODED {
\def\LBCode{[}
\def\EncodedLB{\EscapeChar\LBCode}
\def\UnencodedLB{.}
\def\LB{\EncodedLB}

% |] IS AN ENCODED }
\def\RBCode{]}
\def\EncodedRB{\EscapeChar\RBCode}
\def\UnencodedRB{.}
\def\RB{\EncodedRB}

% |z IS AN ENCODED \dag
\def\DaggerCode{z}
\def\EncodedDagger{\EscapeChar\DaggerCode}
```

```
\def\Dagger{.}% CANNOT \edef \dag

% |Z IS AN ENCODED \ddag
\def\DoubleDaggerCode{Z}
\def\EncodedDoubleDagger{\EscapeChar\DoubleDaggerCode}
\def\DoubleDagger{.}% CANNOT \edef \ddag

% |S IS AN ENCODED \S
\def\SectionSymbolCode{S}
\def\EncodedSectionSymbol{\EscapeChar\SectionSymbolCode}
\def\SectionSymbol{\SaveSectionSymbol}

% |P IS AN ENCODED \P
\def\PilcrowCode{P}
\def\EncodedPilcrow{\EscapeChar\PilcrowCode}
\def\Pilcrow{.}% CANNOT \edef \P

% |E IS AN ENCODED \AE
\def\AEscCode{E}
\def\EncodedAEsc{\EscapeChar\AEscCode}
\def\AEsc{\SaveAEsc}

% |e IS AN ENCODED \ae
\def\aescCode{e}
\def\Encodedaesc{\EscapeChar\aescCode}
\def\aesc{\Saveaesc}

% |O IS AN ENCODED \OE
\def\OEthelCode{O}
\def\EncodedOEthel{\EscapeChar\OEthelCode}
\def\OEthel{\SaveOEthel}

% |o IS AN ENCODED \oe
\def\oethelCode{o}
\def\Encodedoethel{\EscapeChar\oethelCode}
\def\oethel{\Saveoethel}

% |A IS AN ENCODED \AA
\def\AngstromCode{A}
\def\EncodedAngstrom{\EscapeChar\AngstromCode}
\def\Angstrom{\SaveAngstrom}

% |a IS AN ENCODED \aa
\def\angstromCode{a}
\def\Encodedangstrom{\EscapeChar\angstromCode}
\def\angstrom{\Saveangstrom}

% |Q IS AN ENCODED \O
\def\SlashedOCode{Q}
\def\EncodedSlashedO{\EscapeChar\SlashedOCode}
```

```
\def\SlashedO{\SaveSlashedO}

% |q IS AN ENCODED \o
\def\SlashedoCode{q}
\def\EncodedSlashedo{\EscapeChar\SlashedoCode}
\def\Slashedo{\SaveSlashedo}

% |L IS AN ENCODED \L
\def\BarredLCode{L}
\def\EncodedBarredL{\EscapeChar\BarredLCode}
\def\BarredL{\SaveBarredL}

% |l IS AN ENCODED \l
\def\BarredlCode{l}
\def\EncodedBarredl{\EscapeChar\BarredlCode}
\def\Barredl{\SaveBarredl}

% |s IS AN ENCODED \ss
\def\EszettCode{s}
\def\EncodedEszett{\EscapeChar\EszettCode}
\def\Eszett{\SaveEszett}

\newcounter{@letterindex}
\newcounter{@@letterindex}
\newcounter{@@@letterindex}
\newcounter{@wordindex}
\newcounter{@iargc}
\newcounter{@gobblesize}
\newcounter{@maxrotation}
\newcounter{@stringsize}
\newcounter{@@stringsize}
\newcounter{@revisedstringsize}
\newcounter{@gobbleindex}
\newcounter{@charsfound}
\newcounter{@alph}
\newcounter{@alphaindex}
\newcounter{@capstrigger}
\newcounter{@fromindex}
\newcounter{@toindex}
\newcounter{@previousindex}
\newcounter{@flag}
\newcounter{@matchloc}
\newcounter{@matchend}
\newcounter{@matchsize}
\newcounter{@matchmax}
\newcounter{@skipped}

%%%%% CONFIGURATION COMMANDS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

`\defaultTreatments` This command can be used to restore the default string treatments, prior to calling

\substring. The default treatments leave all symbol types intact and unaltered.

```
\newcommand\defaultTreatments{%
  \def\EncodingTreatment{v}% <--Set=v to decode special chars (vs. q,e)
  \def\AlphaCapsTreatment{1}% <--Set=1 to retain uppercase (vs. 0,2)
  \def\AlphaTreatment{1}% <--Set=1 to retain lowercase (vs. 0,2)
  \def\PunctuationTreatment{1}% <--Set=1 to retain punctuation (vs. 0)
  \def\NumeralTreatment{1}% <--Set=1 to retain numerals (vs. 0)
  \def\SymbolTreatment{1}% <--Set=1 to retain special chars (vs. 0)
  \def\BlankTreatment{1}% <--Set=1 to retain blanks (vs. 0)
  \def\CapitalizeString{0}%     <--Set=0 for no special action (vs. 1,2)
  \def\SeekBlankSpace{0}%       <--Set=0 for no special action (vs. 1,2)
}
\defaultTreatments
```

\Treatments This command allows the user to specify the desired character class treatments, prior to a call to \substring. Unfortunately for the user, I have specified which character class each symbol belongs to. Therefore, it is not easy if the user decides that he wants a cedilla, for example, to be treated like an alphabetic character rather than a symbol.

```
% QUICK WAY TO SET UP TREATMENTS BY WHICH \rotate HANDLES VARIOUS
% CHARACTERS
\newcommand\Treatments[6]{%
  \def\AlphaCapsTreatment{#1}%  <--Set=0 to remove uppercase
%                                     =1 to retain uppercase
%                                     =2 to change UC to lc
  \def\AlphaTreatment{#2}%      <--Set=0 to remove lowercase
%                                     =1 to retain lowercase
%                                     =2 to change lc to UC
  \def\PunctuationTreatment{#3}%<--Set=0 to remove punctuation
%                                     =1 to retain punctuation
  \def\NumeralTreatment{#4}%    <--Set=0 to remove numerals
%                                     =1 to retain numerals
  \def\SymbolTreatment{#5}%     <--Set=0 to remove special chars
%                                     =1 to retain special chars
  \def\BlankTreatment{#6}%      <--Set=0 to remove blanks
%                                     =1 to retain blanks
}
```

\+ This command (\+) is used to enact the stringstrings encoding. Key symbols are redefined, and any \edef which occurs while this command is active will adopt these new definitions.

```
% REENCODE MULTIBYTE SYMBOLS USING THE stringstrings ENCODING METHOD
\newcommand\+{%
  \def\${\EncodedDollar}%
  \def~{\EncodedBlankSpace}%
  \def\^{\EncodedCircumflex}%
  \def\~{\EncodedTilde}%
```

```
  \def\"{\EncodedUmlaut}%
  \def\‘{\EncodedGrave}%
  \def\’{\EncodedAcute}%
  \def\={\EncodedMacron}%
  \def\.{\EncodedOverdot}%
  \def\u{\EncodedBreve}%
  \def\v{\EncodedCaron}%
  \def\H{\EncodedDoubleAcute}%
  \def\c{\EncodedCedilla}%
  \def\d{\EncodedUnderdot}%
  \def\t{\EncodedArchJoin}%
  \def\b{\EncodedLineUnder}%
  \def\copyright{\EncodedCopyright}%
  \def\pounds{\EncodedPounds}%
  \def\{{\EncodedLeftBrace}%
  \def\}{\EncodedRightBrace}%
  \def\_{\EncodedUnderscore}%
  \def\dag{\EncodedDagger}%
  \def\ddag{\EncodedDoubleDagger}%
  \def\S{\EncodedSectionSymbol}%
  \def\P{\EncodedPilcrow}%
  \def\AE{\EncodedAEsc}%
  \def\ae{\Encodedaesc}%
  \def\OE{\EncodedOEthel}%
  \def\oe{\Encodedoethel}%
  \def\AA{\EncodedAngstrom}%
  \def\aa{\Encodedangstrom}%
  \def\O{\EncodedSlashedO}%
  \def\o{\EncodedSlashedo}%
  \def\L{\EncodedBarredL}%
  \def\l{\EncodedBarredl}%
  \def\ss{\EncodedEszett}%
}
```

`\?` The command `\?` reverts the character encodings back to the standard LaTeX definitions. The command effectively undoes a previously enacted `\+`.

```
% WHEN TASK IS DONE, REVERT ENCODING TO STANDARD ENCODING METHOD
\newcommand\?{%
  \let\$\SaveDollar%
  \let~\SaveHardspace%
  \let\^\SaveCircumflex%
  \let\~\SaveTilde%
  \let\"\SaveUmlaut%
  \let\‘\SaveGrave%
  \let\’\SaveAcute%
  \let\=\SaveMacron%
  \let\.\SaveOverdot%
  \let\u\SaveBreve%
  \let\v\SaveCaron%
```

```
  \let\H\SaveDoubleAcute%
  \let\c\SaveCedilla%
  \let\d\SaveUnderdot%
  \let\t\SaveArchJoin%
  \let\b\SaveLineUnder%
  \let\copyright\SaveCopyright%
  \let\pounds\SavePounds%
  \let\{\SaveLeftBrace%
  \let\}\SaveRightBrace%
  \let\_\SaveUnderscore%
  \let\dag\SaveDagger%
  \let\ddag\SaveDoubleDagger%
  \let\S\SaveSectionSymbol%
  \let\P\SavePilcrow%
  \let\AE\SaveAEsc%
  \let\ae\Saveaesc%
  \let\OE\SaveOEthel%
  \let\oe\Saveoethel%
  \let\AA\SaveAngstrom%
  \let\aa\Saveangstrom%
  \let\O\SaveSlashedO%
  \let\o\SaveSlashedo%
  \let\L\SaveBarredL%
  \let\l\SaveBarredl%
  \let\ss\SaveEszett%
}
%%%%% COMMANDS TO MANIPULATE STRINGS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

In the next group of commands, the result is always stored in an expandable string, `\thestring`. Expandable means that `\thestring` can be put into a subsequent `\edef{}` command. Additionally, the optional first argument can be used to cause three actions (verbose, encoded, or quiet):

`=v` `\thestring` is decoded (final result); print it immediately (default)

`=e` `\thestring` is encoded (intermediate result); don't print it

`=q` `\thestring` is decoded (final result), but don't print it

`\substring` The command `\substring` is the brains of this package... It is used to acquire a substring from a given string, along with performing specified character manipulations along the way. Its strategy is fundamental to the stringstrings package: sequentially rotate the 1st character of the string to the end of the string, until the desired substring resides at end of rotated string. Then, gobble up the leading part of string until only the desired substring is left.

```
\newcommand\substring[4][v]{\+%
```

Obtain the string length of the string to be manipulated and store it in `@stringsize`.

```
\@getstringlength{#2}{@stringsize}%
```

First, `\@decodepointer` is used to convert indirect references like `$` and `$-3` into integers.

```
\@decodepointer{#3}%
\setcounter{@fromindex}{\fromtoindex}%
\@decodepointer{#4}%
\setcounter{@toindex}{\fromtoindex}%
```

Determine the number of characters to rotate to the end of the string and the number of characters to then gobble from it, in order to leave the desired substring.

```
\setcounter{@gobblesize}{\value{@stringsize}}%
\ifthenelse{\value{@toindex} > \value{@stringsize}}%
  {\setcounter{@maxrotation}{\value{@stringsize}}}%
  {\setcounter{@maxrotation}{\value{@toindex}}}%
\addtocounter{@gobblesize}{-\value{@maxrotation}}%
\addtocounter{@gobblesize}{\value{@fromindex}}%
\addtocounter{@gobblesize}{-1}%
```

Prepare for the string rotation by initializing counters, setting the targeted string into the working variable, `\rotatingword`, and set the encoding treatment specified.

```
\setcounter{@letterindex}{0}%
\edef\rotatingword{#2}%
\def\EncodingTreatment{#1}%
```

If capitalization (first character of string or of each word) was specified, the trigger for 1st-character capitalization will be set. However, the treatments for the alphabetic characters for the remainder of the string must be saved and reinstituted after the first character is capitalized.

```
\if 0\CapitalizeString%
%   DO NOT SET CAPITALIZE TRIGGER FOR FIRST CHARACTER
    \setcounter{@capstrigger}{0}%
\else
%   SAVE CERTAIN TREATMENTS FOR LATER RESTORATION
    \let\SaveAlphaTreatment\AlphaTreatment%
    \let\SaveAlphaCapsTreatment\AlphaCapsTreatment%
%   SET CAPITALIZE TRIGGER FOR FIRST CHARACTER
    \setcounter{@capstrigger}{1}%
    \@forcecapson%
\fi
```

The command `\@defineactions` looks at the defined treatments and specifies how each of the stringstrings encoded characters should be handled (*i.e.*, left alone, removed, modified, *etc.*).

```
\@defineactions%
```

Here begins the primary loop of `\substring` in which characters of `\rotatingword` are successively moved (and possibly manipulated) from the first character of the string to the last. `@letterindex` is the running index defining how many characters have been operated on.

```
  \whiledo{\value{@letterindex} < \value{@maxrotation}}{%
    \addtocounter{@letterindex}{1}%
```

When `\CapitalizeString` equals 1, only the first character of the string is capitalized. When it equals 2, every word in the string is capitalized. When equal to 2, this bit of code looks for the blankspace that follows the end of a word, and uses it to reset the capitalization trigger for the next non-blank character.

```
%   IF NEXT CHARACTER BLANK WHILE \CapitalizeString=2,
%   SET OR KEEP ALIVE TRIGGER.
    \if 2\CapitalizeString%
      \isnextbyte[q]{\EncodedBlankSpace}{\rotatingword}%
      \if F\theresult\isnextbyte[q]{\BlankSpace}{\rotatingword}\fi%
      \if T\theresult%
        \if 0\arabic{@capstrigger}%
          \@forcecapson%
          \@defineactions%
        \fi
        \setcounter{@capstrigger}{2}%
      \fi
    \fi
```

Is the next character an encoded symbol? If it is a normal character, simply rotate it to the end of the string. If it is an encoded symbol however, its treatment will depend on whether it will be gobbled away or end up in the final substring. If it will be gobbled away, leave it encoded, because the gobbling routine knows how to gobble encoded characters. If it will end up in the substring, manipulate it according to the encoding rules set in `\@defineactions` and rotate it.

```
%   CHECK IF NEXT CHARACTER IS A SYMBOL
    \isnextbyte[q]{\EscapeChar}{\rotatingword}%
    \ifthenelse{\value{@letterindex} < \value{@fromindex}}%
      {%
%     THIS CHARACTER WILL EVENTUALLY BE GOBBLED
      \if T\theresult%
%       ROTATE THE ESCAPE CHARACTER, WHICH WILL LEAVE THE SYMBOL ENCODED
%       FOR PROPER GOBBLING (ESCAPE CHARACTER DOESN'T COUNT AS A LETTER)
        \edef\rotatingword{\rotate{\rotatingword}}%
        \addtocounter{@letterindex}{-1}%
      \else
%       NORMAL CHARACTER OR SYMBOL CODE... ROTATE IT
        \edef\rotatingword{\rotate{\rotatingword}}%
      \fi
      }%
```

```
      {%
%     THIS CHARACTER WILL EVENTUALLY MAKE IT INTO SUBSTRING
      \if T\theresult%
%       ROTATE THE SYMBOL USING DEFINED TREATMENT RULES
        \edef\rotatingword{\ESCrotate{\expandafter\@gobble\rotatingword}}%
      \else
%       NORMAL CHARACTER... ROTATE IT
        \edef\rotatingword{\rotate{\rotatingword}}%
      \fi
      }%
```

Here, the capitalization trigger persistently tries to turn itself off with each loop through the string rotation. Only if the earlier code found the rotation to be pointing to the blank character(s) between words while `\CapitalizeString` equals 2 will the trigger be prevented from extinguishing itself.

```
%   DECREMENT CAPITALIZATION TRIGGER TOWARDS 0, EVERY TIME THROUGH LOOP
    \if 0\arabic{@capstrigger}%
    \else
      \addtocounter{@capstrigger}{-1}%
      \if 0\arabic{@capstrigger}\@relaxcapson\fi
    \fi
```

In addition to the standard `\substring` calls in which fixed substring limits are specified (which in turn fixes the number of character rotations to be executed), some stringstrings commands want the rotations to continue until a blankspace is located. This bit of code looks for that blank space, if that was the option requested. Once found, the rotation will stop. However, depending on the value of `\SeekBlankSpace`, the remainder of the string may either be retained or discarded.

```
%   IF SOUGHT SPACE IS FOUND, END ROTATION OF STRING
    \if 0\SeekBlankSpace\else
      \isnextbyte[q]{\EncodedBlankSpace}{\rotatingword}%
      \if F\theresult\isnextbyte[q]{\BlankSpace}{\rotatingword}\fi%
      \if T\theresult%
        \if 1\SeekBlankSpace%
%         STOP ROTATION, KEEP REMAINDER OF STRING
          \setcounter{@maxrotation}{\value{@letterindex}}%
        \else
%         STOP ROTATION, THROW AWAY REMAINDER OF STRING
          \addtocounter{@gobblesize}{\value{@maxrotation}}%
          \setcounter{@maxrotation}{\value{@letterindex}}%
          \addtocounter{@gobblesize}{-\value{@maxrotation}}%
        \fi
      \fi
    \fi
  }%
```

The loop has ended.
Gobble up the first `@gobblesize` characters (not bytes!) of the string, which

should leave the desired substring as the remainder. If the mode is verbose, print out the resulting substring.

```
% GOBBLE AWAY THAT PART OF STRING THAT ISN'T PART OF REQUESTED SUBSTRING
  \@gobblearg{\rotatingword}{\arabic{@gobblesize}}%
  \edef\thestring{\gobbledword}%
  \if v#1\thestring\fi%
\?}
```

Many of the following commands are self-expanatory. The recipe they follow is to use `\Treatments` to specify how different character classes are to be manipulated, and then to call upon `\substring` to effect the desired manipulation. Treatments are typically re-defaulted at the conclusion of the command, which is why the user, if desiring special treatments, should specify those treatments immediately before a call to `\substring`.

`\caseupper`

```
% Convert Lower to Uppercase; retain all symbols, numerals,
% punctuation, and blanks.
\newcommand\caseupper[2][v]{%
  \Treatments{1}{2}{1}{1}{1}{1}%
  \substring[#1]{#2}{1}{\@MAXSTRINGSIZE}%
  \defaultTreatments%
}
```

`\caselower`

```
% Convert Upper to Lowercase; retain all symbols, numerals,
% punctuation, and blanks.
\newcommand\caselower[2][v]{%
  \Treatments{2}{1}{1}{1}{1}{1}%
  \substring[#1]{#2}{1}{\@MAXSTRINGSIZE}%
  \defaultTreatments%
}
```

`\solelyuppercase`

```
% Convert Lower to Uppercase; discard symbols, numerals, and
% punctuation, but keep blanks.
\newcommand\solelyuppercase[2][v]{%
  \Treatments{1}{2}{0}{0}{0}{1}%
  \substring[#1]{#2}{1}{\@MAXSTRINGSIZE}%
  \defaultTreatments%
}
```

`\solelylowercase`

```
% Convert Upper to Lowercase; discard symbols, numerals, and
% punctuation, but keep blanks.
```

```
\newcommand\solelylowercase[2][v]{%
  \Treatments{2}{1}{0}{0}{0}{1}%
  \substring[#1]{#2}{1}{\@MAXSTRINGSIZE}%
  \defaultTreatments%
}
```

\changecase

```
% Convert Lower to Uppercase & Upper to Lower; retain all symbols, numerals,
% punctuation, and blanks.
\newcommand\changecase[2][v]{%
  \Treatments{2}{2}{1}{1}{1}{1}%
  \substring[#1]{#2}{1}{\@MAXSTRINGSIZE}%
  \defaultTreatments%
}
```

\noblanks

```
% Remove blanks; retain all else.
\newcommand\noblanks[2][v]{%
  \Treatments{1}{1}{1}{1}{1}{0}%
  \substring[#1]{#2}{1}{\@MAXSTRINGSIZE}%
  \defaultTreatments%
}
```

\nosymbolsnumerals

```
% Retain case; discard symbols & numerals; retain
% punctuation & blanks.
\newcommand\nosymbolsnumerals[2][v]{%
  \Treatments{1}{1}{1}{0}{0}{1}%
  \substring[#1]{#2}{1}{\@MAXSTRINGSIZE}%
  \defaultTreatments%
}
```

\alphabetic

```
% Retain case; discard symbols, numerals &
% punctuation; retain blanks.
\newcommand\alphabetic[2][v]{%
  \Treatments{1}{1}{0}{0}{0}{1}%
  \substring[#1]{#2}{1}{\@MAXSTRINGSIZE}%
  \defaultTreatments%
}
```

\capitalize The command \CapitalizeString is not set by \Treatments, but only in \capitalize or in \capitalizewords.

```
% Capitalize first character of string,
\newcommand\capitalize[2][v]{%
```

```
  \defaultTreatments%
  \def\CapitalizeString{1}%
  \substring[#1]{#2}{1}{\@MAXSTRINGSIZE}%
  \def\CapitalizeString{0}%
}
```

\capitalizewords

```
% Capitalize first character of each word in string,
\newcommand\capitalizewords[2][v]{%
  \defaultTreatments%
  \def\CapitalizeString{2}%
  \substring[#1]{#2}{1}{\@MAXSTRINGSIZE}%
  \def\CapitalizeString{0}%
}
```

\reversestring Reverses a string from back to front. To do this, a loop is set up in which characters are grabbed one at a time from the end of the given string, working towards the beginning of the string. The grabbed characters are concatenated onto the end of the working string, \@reversedstring. By the time the loop is complete \@reversedstring fully represents the reversed string. The result is placed into \thestring.

```
% REVERSES SEQUENCE OF BYTES IN STRING
\newcommand\reversestring[2][v]{%
  \def\@reversedstring{}%
  \+\@getstringlength{#2}{@@stringsize}\?%
  \setcounter{@@@letterindex}{\the@@stringsize}%
  \whiledo{\the@@@letterindex > 0}{%
    \if e#1%
      \substring[e]{#2}{\the@@@letterindex}{\the@@@letterindex}%
    \else
      \substring[q]{#2}{\the@@@letterindex}{\the@@@letterindex}%
    \fi
    \edef\@reversedstring{\@reversedstring\thestring}%
    \addtocounter{@@@letterindex}{-1}%
  }%
  \edef\thestring{\@reversedstring}%
  \if v#1\thestring\fi%
}
```

\convertchar Takes a string, a replaces each occurance of a specified character with a replacement string. The only complexity in the logic is that a separate replacement algorithm exists depending on whether the specified character to be replaced is a normal character or an encoded character.

```
% TAKES A STARTING STRING #2 AND SUBSTITUTES A SPECIFIED STRING #4
% FOR EVERY OCCURANCE OF A PARTICULAR GIVEN CHARACTER #3.  THE
% CHARACTER TO BE CONVERTED MAY BE EITHER A PLAIN CHARACTER OR
```

```
% AN ENCODABLE SYMBOL.
\newcommand\convertchar[4][v]{%
  \+%
  \edef\encodedstring{#2}
  \edef\encodedfromarg{#3}%
  \edef\encodedtoarg{#4}%
  \?%
  \isnextbyte[q]{\EscapeChar}{\encodedfromarg}
  \if F\theresult%
%   PLAIN "FROM" ARGUMENT
    \@convertbytetostring[#1]{\encodedstring}{#3}{\encodedtoarg}%
  \else
%   ENCODABLE "FROM" ARGUMENT
    \@convertsymboltostring[#1]{\encodedstring}%
       {\expandafter\@gobble\encodedfromarg}{\encodedtoarg}%
  \fi
}

```

`\rotateword` Moves first word of given string #2 to end of string, including leading and trailing blank spaces.

```
\newcommand\rotateword[2][v]{%
  \+\edef\thestring{#2}\?%
```

Rotate leading blank spaces to end of string

```
  \@treatleadingspaces[e]{\thestring}{}%
```

Define end-of-rotate condition for `\substring` as next blank space

```
  \def\SeekBlankSpace{1}%
```

Leave rotated characters alone

```
  \Treatments{1}{1}{1}{1}{1}{1}%
```

Rotate to the next blank space or the end of string, whichever comes first.

```
  \substring[e]{\thestring}{1}{\@MAXSTRINGSIZE}%
```

Rotate trailing spaces.

```
  \@treatleadingspaces[#1]{\thestring}{}%
  \defaultTreatments%
}
```

`\removeword` Remove the first word of given string #2, including leading and trailing spaces. Note that logic is identical to `\rotateword`, except that affected spaces and characters are removed instead of being rotated.

```
\newcommand\removeword[2][v]{%
  \+\edef\thestring{#2}\?%
```

The {x} final argument indicates to delete leading spaces.

```
  \@treatleadingspaces[e]{\thestring}{x}%
  \def\SeekBlankSpace{1}%
```

The Treatments are specified to remove all characters.

```
  \Treatments{0}{0}{0}{0}{0}{0}%
  \substring[e]{\thestring}{1}{\@MAXSTRINGSIZE}%
```

Trailing spaces are also deleted.

```
  \@treatleadingspaces[#1]{\thestring}{x}%
  \defaultTreatments%
}
```

`\getnextword` A special case of `\getaword`, where word-to-get is specified as "1".

```
% GETS NEXT WORD FROM STRING #2.
% NOTE: ROTATES BACK TO BEGINNING, AFTER STRING OTHERWISE EXHAUSTED
\newcommand\getnextword[2][v]{%
  \getaword[#1]{#2}{1}%
}
```

`\getaword` Obtain a specified word number (#3) from string #2. Logic: rotate leading spaces to end of string; then loop #3 – 1 times through `\rotateword`. Finally, get next word.

```
% GETS WORD #3 FROM STRING #2.
% NOTE: ROTATES BACK TO BEGINNING, AFTER STRING OTHERWISE EXHAUSTED
\newcommand\getaword[3][v]{%
  \setcounter{@wordindex}{1}%
  \+\edef\thestring{#2}\?%
  \@treatleadingspaces[e]{\thestring}{}%
  \whiledo{\value{@wordindex} < #3}{%
    \rotateword[e]{\thestring}%
    \addtocounter{@wordindex}{1}%
  }%
  \@getnextword[#1]{\thestring}%
}
```

`\rotateleadingspaces` Rotate leading spaces of string #2 to the end of string.

```
\newcommand\rotateleadingspaces[2][v]{%
  \@treatleadingspaces[#1]{#2}{}%
}
```

`\removeleadingspaces` Remove leading spaces from string #2.

```
\newcommand\removeleadingspaces[2][v]{%
  \@treatleadingspaces[#1]{#2}{x}%
}
```

\stringencode

```
% ENCODE STRING; UNLIKE OTHER COMMANDS, DEFAULT IS NO PRINT
\newcommand\stringencode[2][e]{%
  \defaultTreatments%
  \substring[#1]{#2}{1}{\@MAXSTRINGSIZE}%
}
```

\stringdecode

```
% DECODE STRING
\newcommand\stringdecode[2][v]{%
  \defaultTreatments%
  \substring[#1]{#2}{1}{\@MAXSTRINGSIZE}%
}
```

\gobblechar Remove first character (not byte!) from string #2. Unlike just about all other stringstrings commands, result is retokenized and not expanded.

```
% SINGLE-CHARACTER VERSION OF \gobblechars.  IN THIS CASE, TWO-BYTE
% ESCAPE SEQUENCES, WHEN ENCOUNTERED, COUNT AS A SINGLE GOBBLE.
\newcommand\gobblechar[2][q]{\+%
  \@gobblearg{#2}{1}%
  \?\retokenize[#1]{\gobbledword}%
}
```

\gobblechars Remove first #3 characters (not bytes!) from string #2. Unlike just about all other stringstrings commands, result is retokenized and not expanded.

```
% USER CALLABLE VERSION OF \@gobblearg.  TURNS ON REENCODING.
% GOBBLE FIRST #3 CHARACTERS FROM STRING #2. TWO-BYTE
% ESCAPE SEQUENCES, WHEN ENCOUNTERED, COUNT AS A SINGLE GOBBLE.
\newcommand\gobblechars[3][q]{\+%
  \@gobblearg{#2}{#3}%
  \?\retokenize[#1]{\gobbledword}%
}
```

\retokenize One of the key stringstrings routines that provides several indispensible functions. Its function is to take an encoded string #2 that has been given, and repopulate the string with its LaTeX tokens in a `\def` form (not an expanded `\edef` form). It is useful if required to operate on a string outside of the stringstrings library routines, following a stringstrings manipulation. It is also useful to display certain tokens which cannot be manipulated in expanded form. See Table 1 for a list of tokens that will only work when the resulting string is retokenized (and not expanded).

Logic: Loop through each character of given string #2. Each successive character of the string is retokenized as `\inexttoken`, `\iinexttoken`, `\iiinexttoken`, `\ivnexttoken`, *etc.*, respectively. Then a series of strings are formed as

```
\def\buildtoken{}
\def\buildtokeni{\builtoken\inexttoken}
\def\buildtokenii{\builtokeni\iinexttoken}
\def\buildtokeniii{\builtokenii\iiinexttoken}
\def\buildtokeniv{\builtokeniii\ivnexttoken}
```

The last in the sequence of `\builtoken...` strings (renamed `\buildtokenq`) is the retokenized version of string #2.

```
% CONVERTS ENCODED STRING BACK INTO TOKENIZED FORM (i.e., def'ED).
\newcommand\retokenize[2][q]{\+%
  \edef\@svstring{#2}%
  \edef\buildtoken{}%
  \@getstringlength{#2}{@@stringsize}\?%
  \setcounter{@@letterindex}{0}%
  \whiledo{\the@@letterindex < \the@@stringsize}{%
    \setcounter{@previousindex}{\the@@letterindex}%
    \addtocounter{@@letterindex}{1}%
    \substring[e]{\@svstring}{\the@@letterindex}{\the@@letterindex}%
    \@retokenizechar{\thestring}{\roman{@@letterindex}nexttoken}%
    \expandafter\def\csname buildtoken\roman{@@letterindex}%
     \expandafter\endcsname\expandafter%
      {\csname buildtoken\roman{@previousindex}\expandafter\endcsname%
       \csname\roman{@@letterindex}nexttoken\endcsname}%
  }%
  \expandafter\def\expandafter\buildtokenq\expandafter{%
    \csname buildtoken\roman{@@letterindex}\endcsname}%
  \def\thestring{\buildtokenq}%
  \if v#1\thestring\fi
}

% %%%%% COMMANDS TO EXTRACT STRING INFORMATION %%%%%%%%%%%%%%%%%%%%%%%%%
```

The following group of commands extract information about a string, and store the result in a string called `\theresult`. Since the result is not a substring, a *mode* of `[e]` carries no meaning. Only `[v]` and `[q]` modes apply here.

`\stringlength` Returns the length of the given string in *characters*, not *bytes*.

```
% USER CALLABLE VERSION of \@getstringlength:
% GET'S STRING LENGTH OF [#2], PUTS RESULT IN \theresult.  PRINTS RESULT
% UNLESS IN QUIET [q] MODE.
\newcommand\stringlength[2][v]{\+%
  \@getstringlength{#2}{@@stringsize}%
  \edef\theresult{\arabic{@@stringsize}}%
  \if v#1\theresult\fi%
\?}
```

`\findchars` Find number of occurances of character `#3` in string `#2`.

```
% CHECKS TO SEE IF THE CHARACTER [#3] APPEARS ANYWHERE IN STRING [#2].
% THE NUMBER OF OCCURANCES IS PRINTED OUT, EXCEPT WHEN [#1]=q, QUIET
% MODE.  RESULT IS ALSO STORED IN \theresult .  TO FIND SPACES, ARG3
% SHOULD BE SET TO {~}, NOT { }.
\newcommand\findchars[3][v]{\+%
  \@getstringlength{#2}{@@stringsize}%
  \setcounter{@charsfound}{0}%
  \setcounter{@@letterindex}{0}%
```

Loop through each character of #2.

```
  \whiledo{\value{@@letterindex} < \value{@@stringsize}}{%
    \addtocounter{@@letterindex}{1}%
```

Test if the @@letterindex character of string #2 equals #3. If so, add to tally.

```
    \testmatchingchar{#2}{\arabic{@@letterindex}}{#3}%
    \ifmatchingchar\addtocounter{@charsfound}{1}\fi
  }%
  \edef\theresult{\arabic{@charsfound}}%
  \if q#1\else\theresult\fi%
\?}
```

\whereischar Similar to \findchars, but instead finds first occurance of match character #3 within #2 and returns its location within #2.

```
% CHECKS TO FIND LOCATION OF FIRST OCCURANCE OF [#3] IN STRING [#2].
% THE LOCATION IS PRINTED OUT, EXCEPT WHEN [#1]=q, QUIET
% MODE.  RESULT IS ALSO STORED IN \theresult .  TO FIND SPACES, ARG3
% SHOULD BE SET TO {~}, NOT { }.
\newcommand\whereischar[3][v]{\+%
  \@getstringlength{#2}{@@stringsize}%
  \edef\@theresult{0}%
  \setcounter{@@letterindex}{0}%
```

Loop through characters of #2 sequentially.

```
  \whiledo{\value{@@letterindex} < \value{@@stringsize}}{%
    \addtocounter{@@letterindex}{1}%
```

Look for match. If found, save character-location index, and reset loop index to break from loop.

```
    \testmatchingchar{#2}{\arabic{@@letterindex}}{#3}%
    \ifmatchingchar%
      \edef\@theresult{\the@@letterindex}%
      \setcounter{@@letterindex}{\the@@stringsize}%
    \fi
  }%
  \edef\theresult{\@theresult}%
  \if q#1\else\theresult\fi%
\?}
```

`\whereisword` Finds location of specified word (`#3`) in string `#2`.

```
% LIKE \whereischar, EXCEPT FOR WORDS
\newcommand\whereisword[3][v]{\+%
  \setcounter{@skipped}{0}%
```

`\@@@@teststring` initially contains `#2`. As false alarms are located, the string will be redefined to lop off initial characters of string.

```
  \edef\@@@@teststring{#2}%
  \edef\@matchstring{#3}%
  \@getstringlength{#2}{@@stringsize}%
  \@getstringlength{#3}{@matchsize}%
  \setcounter{@matchmax}{\the@@stringsize}%
  \addtocounter{@matchmax}{-\the@matchsize}%
  \addtocounter{@matchmax}{1}%
  \setcounter{@flag}{0}%
```

Define `\matchchar` as the first character of the match string (`#3`).

```
  \substring[e]{#3}{1}{1}%
  \edef\matchchar{\thestring}%
  \whiledo{\the@flag = 0}{%
```

Look for first character of match string within `\@@@@teststring`.

```
    \whereischar[q]{\@@@@teststring}{\matchchar}%
    \setcounter{@matchloc}{\theresult}%
    \ifthenelse{\equal{0}{\value{@matchloc}}}%
```

If none found, we are done.

```
    {%
      \setcounter{@flag}{1}%
    }%
```

If `\matchchar` is found, must determine if it is the beginning of the match string, or just an extraneous match (*i.e.*, false alarm). Extract substring of `\@@@@teststring`, of a size equal to the match string. Compare this extracted string with the match string.

```
    {%
      \setcounter{@matchend}{\theresult}%
      \addtocounter{@matchend}{\value{@matchsize}}%
      \addtocounter{@matchend}{-1}%
      \substring[e]{\@@@@teststring}{\the@matchloc}{\the@matchend}%
      \ifthenelse{\equal{\thestring}{\@matchstring}}%
```

Found a match! Save the match location

```
      {%
        \setcounter{@flag}{1}%
        \addtocounter{@matchloc}{\the@skipped}%
        \edef\theresult{\the@matchloc}%
```

```
759       }%
```

False alarm. Determine if lopping off the leading characters of `\@@@@teststring` (to discard the false-alarm occurance) is feasible. If lopping would take one past the end of the string, then no match is possible. If lopping permissible, redefine the string `\@@@@teststring`, keeping track of the total number of lopped-off characters in the counter `@skipped`.

```
760       {%
761         \addtocounter{@skipped}{\the@matchloc}%
762         \addtocounter{@matchloc}{1}%
763         \ifthenelse{\value{@matchloc} > \value{@matchmax}}%
764         {%
765           \setcounter{@flag}{1}%
766           \edef\theresult{0}%
767         }%
768         {%
769           \substring[e]{\@@@@teststring}{\the@matchloc}{\@MAXSTRINGSIZE}%
770           \edef\@@@@teststring{\thestring}%
771         }%
772       }%
773     }%
774   }%
775   \if q#1\else\theresult\fi%
776 \?}
```

`\findwords` Finds the number of occurances of a word within the provided string

```
777 % LIKE \findchar, EXCEPT FOR WORDS
778 \newcommand\findwords[3][v]{%
779   \+\edef\@@teststring{#2}\?%
780   \edef\@@@teststring{\@@teststring}%
781   \setcounter{@charsfound}{0}%
782   \whiledo{\the@charsfound > -1}{%
```

Seek occurance of `#3` in the string to be tested

```
783     \whereisword[q]{\@@teststring}{#3}%
784     \setcounter{@matchloc}{\theresult}%
785     \ifthenelse{\the@matchloc = 0}%
786       {%
```

None found. Break from loop.

```
787         \edef\theresult{\the@charsfound}%
788         \setcounter{@charsfound}{-1}%
789       }%
790       {%
```

Found. Increment count.

```
791         \addtocounter{@charsfound}{1}%
```

```
        \addtocounter{@matchloc}{\the@matchsize}%
        \ifthenelse{\the@matchloc > \the@@stringsize}%
        {%
```

This "find" takes us to the end-of-string. Break from loop now.

```
          \edef\theresult{\the@charsfound}%
          \setcounter{@charsfound}{-1}%
        }%
        {%
```

More string to search. Lop off what has been searched from string to be tested, and re-loop for next search.

```
          \substring[e]{\@@@teststring}{\the@matchloc}{\@MAXSTRINGSIZE}%
          \edef\@@teststring{\thestring}%
          \edef\@@@teststring{\@@teststring}%
        }%
      }%
  }%
  \if q#1\else\theresult\fi%
}
```

`\convertword` Takes a string, a replaces each occurance of a specified string with a replacement string.

```
\newcounter{@@matchloc}
% LIKE \convertchar, EXCEPT FOR WORDS
\newcommand\convertword[4][v]{%
  \+\edef\@@teststring{#2}%
  \edef\@fromstring{#3}%
  \edef\@tostring{#4}\?%
  \edef\@@@teststring{\@@teststring}%
  \def\@buildfront{}%
  \edef\@buildstring{\@@teststring}%
  \setcounter{@charsfound}{0}%
  \whiledo{\the@charsfound > -1}{%
```

Seek occurance of `\@fromstring` in larger `\@@teststring`

```
\whereisword[q]{\@@teststring}{\@fromstring}%
    \setcounter{@matchloc}{\theresult}%
    \ifthenelse{\the@matchloc = 0}%
      {%
```

Not found. Done.

```
        \setcounter{@charsfound}{-1}%
      }%
      {%
```

Potential matchstring.

```
        \addtocounter{@charsfound}{1}%
```

Grab current test string from beginning to point just prior to potential match.

```
        \addtocounter{@matchloc}{-1}%
        \substring[e]{\@@@teststring}{1}{\the@matchloc}%
```

The string `\@buildfront` is the total original string, with string substitutions, from character 1 to current potential match.

```
        \edef\@buildfront{\@buildfront\thestring}%
```

See if potential matchstring takes us to end-of-string...

```
        \addtocounter{@matchloc}{1}%
        \addtocounter{@matchloc}{\the@matchsize}%
        \ifthenelse{\the@matchloc > \the@@stringsize}%
        {%
```

...if so, then match is last one in string. Tack on replacement string to `\@buildfront` to create final string. Exit.

```
          \setcounter{@charsfound}{-1}%
          \edef\@buildstring{\@buildfront\@tostring}%
        }%
        {%
```

...if not, redefine current teststring to begin at point following the current substitution. Make substitutions to current `\@buildstring` and `\@buildfront`. Loop through logic again on new teststring.

```
          \substring[e]{\@@@teststring}{\the@matchloc}{\@MAXSTRINGSIZE}%
          \edef\@@teststring{\thestring}%
          \edef\@@@teststring{\@@teststring}%
          \edef\@buildstring{\@buildfront\@tostring\@@@teststring}%
          \edef\@buildfront{\@buildfront\@tostring}%
        }%
      }%
  }%
  \substring[#1]{\@buildstring}{1}{\@MAXSTRINGSIZE}%
}
```

`\wordcount` Counts words (space-separated text) in a string. Simply removes one word at a time, counting the words as it goes. With each removal, checks for non-zero string size remaining.

```
% WORD COUNT
\newcommand\wordcount[2][v]{\+%
  \edef\@argv{#2}
  \@getstringlength{\@argv}{@stringsize}
  \setcounter{@iargc}{0}
  \whiledo{\value{@stringsize} > 0}{%
```

```
    \addtocounter{@iargc}{1}%
    \removeword[e]{\@argv}%
    \edef\@argv{\thestring}%
    \@getstringlength{\@argv}{@stringsize}%
  }
  \edef\theresult{\arabic{@iargc}}%
  \if v#1\theresult\fi%
\?}
```

`\getargs` Parse a string of arguments in Unix-like manner. Define `\argv` as `#2`. Grabs leading word from `\argv` and puts it in `\argi`. Increment argument count; remove leading word from `\argv`. Repeat this process, with each new argument being placed in `\argii`, `\argiii`, `\argiv`, *etc.* Continue until size of `\argv` is exhausted.

```
% OBTAINS ARGUMENTS (WORDS) IN #1 ALA UNIX getarg COMMAND
% narg CONTAINS NUMBER OF ARGUMENTS.  ARGUMENTS CONTAINED IN
% argi, argii, argiii, argiv, ETC.
% v mode disabled
\newcommand\getargs[2][q]{\+%
  \if v#1\def\@mode{q}\else\def\@mode{#1}\fi
  \edef\@argv{#2}%
  \@getstringlength{\@argv}{@stringsize}%
  \setcounter{@iargc}{0}%
  \whiledo{\value{@stringsize} > 0}{%
    \addtocounter{@iargc}{1}%
    \getaword[\@mode]{\@argv}{1}%
    \expandafter\edef\csname arg\roman{@iargc}\endcsname{\thestring}%
    \removeword[e]{\@argv}%
    \edef\@argv{\thestring}%
    \@getstringlength{\@argv}{@stringsize}%
  }
  \edef\narg{\arabic{@iargc}}%
\?}

%%%% COMMANDS TO TEST STRINGS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

The following group of commands test for various alphanumeric string conditions.

`\isnextbyte` This routine performs a simple test to determine if the first byte of string `#3` matches the byte given by `#2`. The only problem is that the test can produce a false negative if the first byte of the test string equals the match byte and the second byte of the test string equals the `SignalChar` (defined below).

To resolve this possibility, the test is performed twice with two different values for `\SignalChar`, only one of which can produce a false negative for a given test string. If the two results match, then that result gives the correct answer to the question of whether the first byte of `#3` equals `#2`. If, however, the two results fail to match, then one can assume that one of the results is a false negative, and so

a “true” condition results.

The following two “signal characters,” used for the two tests, can be any two distinct characters. They are used solely by `\isnextbyte`.

```
\def\PrimarySignalChar{@}
\def\SecondarySignalChar{‘}

% \isnextbyte NEEDS TO OPERATE IN RAW (SINGLE BYTE) MODE SO AS TO
% PERFORM TESTS FOR PRESENCE OF \EscapeChar
```

Incidentally, `\isnextbyte` can and is used by stringstrings to detect multi-byte characters in a manner which may also be employed by the user. To do this: First, the string to be tested should be encoded. Then, `\isnextbyte` may be used to check for `\EscapeChar` which is how every multi-byte character will begin its encoding by the stringstrings package. If `\EscapeChar` is detected as the next character, then the string to test may have its leading byte gobbled and the next character (called the Escape Code) may be tested, and compared against the known stringstrings escape codes. The combination of Escape-Character/Escape-Code is how all multi-byte characters are encoded by the stringstrings package.

```
\newcommand\isnextbyte[3][v]{%
```

Here’s the first test...

```
  \let\SignalChar\PrimarySignalChar%
  \edef\@x{\if #2#3\else\SignalChar\fi}%
  \edef\@x{\if \SignalChar\@x F\else T\fi}%
```

...and the second

```
  \let\SignalChar\SecondarySignalChar%
  \edef\@y{\if #2#3\else\SignalChar\fi}%
  \edef\@y{\if \SignalChar\@y F\else T\fi}%
```

If the two tests produced the same result, then a comparison of `\@x\@y` and `\@y\@x` will show it.

```
% BECAUSE THE METHOD ONLY PRODUCES FALSE NEGATIVES, IF RESULTS DON'T
% AGREE FROM USING TWO DIFFERENT SIGNAL CHARACTERS, RESULT MUST BE TRUE.
  \ifthenelse{\equal{\@x\@y}{\@y\@x}}
    {\edef\theresult{\@x}}%
% CORRECT THE FALSE NEGATIVE
    {\edef\theresult{T}}%
  \if q#1\else\theresult\fi
}
```

`\testmatchingchar` This routine checks for a specified match-character within a target string. Unlike `\isnextbyte`, this routine checks for characters (single- or multi-byte) and not just individual bytes. Additionally, rather than testing the match-character against

the first byte of the test-string, the user specifies (through #2) which byte of the test-string should be compared to the match-character.

This routine is not as efficient as \isnextbyte, but much more versatile.

```
% CHECKS TO SEE IF [#2]'th CHARACTER IN STRING [#1] EQUALS [#3]
% RESULT STORED IN BOOLEAN \ifmatchingchar
\newif\ifmatchingchar
\newcommand\testmatchingchar[3]{%
```

Extract desired character from test string

```
  \substring[e]{#1}{#2}{#2}\+%
```

Determine if the match-character is a multi-byte symbol.

```
  \isnextbyte[q]{\EscapeChar}{#3}%
  \if T\theresult%
```

Is the tested character also a multi-byte symbol?

```
    \isnextbyte[q]{\EscapeChar}{\thestring}%
    \if T\theresult%
```

Yes it is... Therefore, compare codes following the escape character

```
      \edef\@testcode{\expandafter\@DiscardNextChar\expandafter{#3}}%
      \edef\@teststring{\@DiscardNextChar{\thestring}}%
      \if \@teststring\@testcode\matchingchartrue\else\matchingcharfalse\fi
    \else
```

No, we are comparing a normal character against a multi-byte symbol (apples and oranges), a false comparison.

```
      \matchingcharfalse%
    \fi
  \else
```

No, we are comparing two normal one-byte characters, not a mult-byte character.

```
    \if \thestring#3\matchingchartrue\else\matchingcharfalse\fi
  \fi
\?}
```

\testcapitalized This routine checks to see if first character of string is capitalized. The only quirk is that the routine must ascertain whether that character is a single-byte character or a multi-byte character.

```
\newif\ifcapitalized
\newcommand\testcapitalized[1]{\+%
  \isnextbyte[q]{\EscapeChar}{#1}%
  \if T\theresult%
    \def\EncodingTreatment{e}%
    \edef\rotatingword{#1}%
```

Rotate the first [multi-byte] character of the string to the end of the string, lowering its case. Store as `\@stringA`.

```
    \def\AlphaCapsTreatment{2}%
    \@defineactions%
    \edef\@stringA{\ESCrotate{\expandafter\@gobble\rotatingword}}%
```

Rotate the first [multi-byte] character of the string to the end of the string, retaining its case. Store as `\@stringB`.

```
    \def\AlphaCapsTreatment{1}%
    \@defineactions%
    \edef\@stringB{\ESCrotate{\expandafter\@gobble\rotatingword}}%
  \else
```

. . . or, if the first character is a normal one-byte character. . . Rotate the first [normal] character of the string to the end of the string, lowering its case. Store as `\@stringA`.

```
    \def\AlphaCapsTreatment{2}%
    \edef\@stringA{\rotate{#1}}%
```

Rotate the first [normal] character of the string to the end of the string, retaining its case. Store as `\@stringB`.

```
    \def\AlphaCapsTreatment{1}%
    \edef\@stringB{\rotate{#1}}%
  \fi
```

Compare strings A and B, to see if changing the case of first letter altered the string

```
  \ifthenelse{\equal{\@stringA}{\@stringB}}%
  {\capitalizedfalse}{\capitalizedtrue}\?%
  \defaultTreatments%
}
```

`\testuncapitalized` This routine is the complement of `\testcapitalized`. The only difference is that the `\@stringA` has its case made upper for the comparison, instead of lowered.

```
\newif\ifuncapitalized
\newcommand\testuncapitalized[1]{\+%
  \isnextbyte[q]{\EscapeChar}{#1}%
  \if T\theresult%
    \def\EncodingTreatment{e}%
    \edef\rotatingword{#1}%
    \def\AlphaTreatment{2}%
    \@defineactions%
    \edef\@stringA{\ESCrotate{\expandafter\@gobble\rotatingword}}%
    \def\AlphaTreatment{1}%
    \@defineactions%
    \edef\@stringB{\ESCrotate{\expandafter\@gobble\rotatingword}}%
```

```
  \else
    \def\AlphaTreatment{2}%
    \edef\@stringA{\rotate{#1}}%
    \def\AlphaTreatment{1}%
    \edef\@stringB{\rotate{#1}}%
  \fi
  \ifthenelse{\equal{\@stringA}{\@stringB}}%
  {\uncapitalizedfalse}{\uncapitalizedtrue}\?%
  \defaultTreatments%
}
```

\testleadingalpha Test if the leading character of the string is alphabetic. This is simply accomplished by checking whether the string is either capitalized or uncapitalized. If non-alphabetic, it will show up as false for both those tests.

```
\newif\ifleadingalpha
\newcommand\testleadingalpha[1]{%
  \testcapitalized{#1}%
  \ifcapitalized
    \leadingalphatrue%
  \else
    \testuncapitalized{#1}%
    \ifuncapitalized
      \leadingalphatrue%
    \else
      \leadingalphafalse%
    \fi
  \fi
}
```

\testuppercase Checks to see if all alphabetic characters in a string are uppercase. Non-alphabetic characters don't affect the result, unless the string is composed solely of nonalphabetic characters, in which case the test results is false.

```
\newif\ifuppercase
\newcommand\testuppercase[1]{%
```

Strip all non-alphabetic characters. Save as \@stringA.

```
  \Treatments{1}{1}{0}{0}{0}{0}%
  \substring[e]{#1}{1}{\@MAXSTRINGSIZE}%
  \edef\@stringA{\thestring}%
```

Lower the case of all uppercase characters in \@stringA. Save as \@stringB. Compare these two strings.

```
  \def\AlphaTreatment{2}%
  \substring[e]{#1}{1}{\@MAXSTRINGSIZE}%
  \edef\@stringB{\thestring}%
  \ifthenelse{\equal{\@stringA}{\@stringB}}%
  {%
```

If the strings are equal, then all the alphabetic characters in the original string were uppercase. Need only check to make sure at least one alphabetic character was present in the original string.

```
    \@getstringlength{\@stringA}{@stringsize}%
    \ifthenelse{\value{@stringsize} = 0}%
    {\uppercasefalse}{\uppercasetrue}%
  }%
```

If strings are not equal, then the alphabetic characters of the original string were not all uppercase. Test false.

```
  {\uppercasefalse}%
  \defaultTreatments%
}
```

`\ifsolelyuppercase` Compare the original string to one made solely uppercase. If they are equal, then the original string was solely uppercase to begin with.

```
\newif\ifsolelyuppercase
\newcommand\testsolelyuppercase[1]{%
  \stringencode{#1}%
  \edef\@stringA{\thestring}%
  \solelyuppercase[e]{#1}%
  \edef\@stringB{\thestring}%
  \ifthenelse{\equal{\@stringA}{\@stringB}}%
  {%
    \noblanks[q]{\@stringA}%
    \@getstringlength{\thestring}{@stringsize}%
    \ifthenelse{\value{@stringsize} = 0}%
    {\solelyuppercasefalse}{\solelyuppercasetrue}%
  }%
  {\solelyuppercasefalse}%
  \defaultTreatments%
}
```

`\testlowercase` This routine is the complement to `\testuppercase`, with corresponding logic.

```
\newif\iflowercase
\newcommand\testlowercase[1]{%
  \Treatments{1}{1}{0}{0}{0}{0}%
  \substring[e]{#1}{1}{\@MAXSTRINGSIZE}%
  \edef\@stringA{\thestring}%
  \def\AlphaCapsTreatment{2}%
  \substring[e]{#1}{1}{\@MAXSTRINGSIZE}%
  \edef\@stringB{\thestring}%
  \ifthenelse{\equal{\@stringA}{\@stringB}}%
  {%
    \@getstringlength{\@stringA}{@stringsize}%
    \ifthenelse{\value{@stringsize} = 0}%
    {\lowercasefalse}{\lowercasetrue}%
```

```
  }%
  {\lowercasefalse}%
  \defaultTreatments%
}
```

`\testsolelylowercase` This routine is the complement to `\testsolelyuppercase`, with corresponding logic.

```
\newif\ifsolelylowercase
\newcommand\testsolelylowercase[1]{%
  \stringencode{#1}%
  \edef\@stringA{\thestring}%
  \solelylowercase[e]{#1}%
  \edef\@stringB{\thestring}%
  \ifthenelse{\equal{\@stringA}{\@stringB}}%
  {%
    \noblanks[q]{\@stringA}%
    \@getstringlength{\thestring}{@stringsize}%
    \ifthenelse{\value{@stringsize} = 0}%
    {\solelylowercasefalse}{\solelylowercasetrue}%
  }%
  {\solelylowercasefalse}%
  \defaultTreatments%
}
```

`\testalphabetic` Comparable to `\testsolelyuppercase` and `\testsolelylowercase` in its logic, this routine tests whether the string is purely alphabetic or not.

```
\newif\ifalphabetic
\newcommand\testalphabetic[1]{%
  \stringencode{#1}%
  \edef\@stringA{\thestring}%
  \alphabetic[e]{#1}%
  \edef\@stringB{\thestring}%
  \ifthenelse{\equal{\@stringA}{\@stringB}}%
  {%
    \noblanks[q]{\@stringA}%
    \@getstringlength{\thestring}{@stringsize}%
    \ifthenelse{\value{@stringsize} = 0}%
    {\alphabeticfalse}{\alphabetictrue}%
  }%
  {\alphabeticfalse}%
  \defaultTreatments%
}

%
%%%%% SUPPORT ROUTINES %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
```

The following routines support the execution of the stringstrings package.

`\rotate` This user-callable routine is purely a placeholder for the underlying service routine.

```
% CALLS ON THE GUTS OF THIS PACKAGE.
\newcommand\rotate[1]{\@rotate{#1}}
```

`\ESCrotate` After the escape character has been ascertained as the next character, this routine operates on the subsequent escape code to rotate the symbol to end of string, in the fashion of macro `\rotate`.

```
\newcommand\ESCrotate[1]{%
  \if\@fromcode#1\@tostring\else
   \if\PipeCode#1\@pipeaction\else
    \if\DollarCode#1\@dollaraction\else
     \if\CaratCode#1\@carataction\else
      \if\CircumflexCode#1\@circumflexaction\else
       \if\TildeCode#1\@tildeaction\else
        \if\UmlautCode#1\@umlautaction\else
         \if\GraveCode#1\@graveaction\else
          \if\AcuteCode#1\@acuteaction\else
           \if\MacronCode#1\@macronaction\else
            \if\OverdotCode#1\@overdotaction\else
             \if\LeftBraceCode#1\@leftbraceaction\else
              \if\RightBraceCode#1\@rightbraceaction\else
               \if\UnderscoreCode#1\@underscoreaction\else
                \if\DaggerCode#1\@daggeraction\else
                 \if\DoubleDaggerCode#1\@doubledaggeraction\else
                  \if\SectionSymbolCode#1\@sectionsymbolaction\else
                   \if\PilcrowCode#1\@pilcrowaction\else
                    \if\LBCode#1\@lbaction\else
                     \if\RBCode#1\@rbaction\else
  \if\BreveCode#1\@breveaction\else
   \if\CaronCode#1\@caronaction\else
    \if\DoubleAcuteCode#1\@doubleacuteaction\else
     \if\CedillaCode#1\@cedillaaction\else
      \if\UnderdotCode#1\@underdotaction\else
       \if\ArchJoinCode#1\@archjoinaction\else
        \if\LineUnderCode#1\@lineunderaction\else
         \if\CopyrightCode#1\@copyrightaction\else
          \if\PoundsCode#1\@poundsaction\else
           \if\AEscCode#1\@AEscaction\else
            \if\aescCode#1\@aescaction\else
             \if\OEthelCode#1\@OEthelaction\else
              \if\oethelCode#1\@oethelaction\else
               \if\AngstromCode#1\@Angstromaction\else
                \if\angstromCode#1\@angstromaction\else
                 \if\SlashedOCode#1\@slashedOaction\else
                  \if\SlashedoCode#1\@slashedoaction\else
                   \if\BarredlCode#1\@barredlaction\else
                    \if\BarredLCode#1\@barredLaction\else
                     \if\EszettCode#1\@eszettaction\else
```

```
                        \expandafter\@gobble#1\undecipherable%
                      \fi
                     \fi
                    \fi
                   \fi
                  \fi
                 \fi
                \fi
               \fi
              \fi
             \fi
            \fi
           \fi
          \fi
         \fi
        \fi
       \fi
      \fi
     \fi
    \fi
   \fi
  \fi
                      \fi
                     \fi
                    \fi
                   \fi
                  \fi
                 \fi
                \fi
               \fi
              \fi
             \fi
            \fi
           \fi
          \fi
         \fi
        \fi
       \fi
      \fi
     \fi
    \fi
   \fi
  \fi
}
```

`\@getnextword` A low-level routine designed to extract the next [space-delimited] word of the primary argument. It has several quirks: if the passed string has one leading space, it is included as part of next word. If it has two leading [hard]spaces, the 2nd hard space *is* the next word. Using the higher-level `\getnextword` deals automatically with these abberant possibilities.

```
\newcommand\@getnextword[2][v]{%
```

```
  \defaultTreatments%
  \def\SeekBlankSpace{2}%
  \substring[#1]{#2}{1}{\@MAXSTRINGSIZE}%
  \def\SeekBlankSpace{0}%
}
```

`\@retokenizechar` This command is the guts of the retokenize command. It grabs the next character from string `#1` and assigns it to a unique token whose name is created using from the string `#2`. The command has two primary `\if` branches. The branch is taken if the character is a special escape-sequence character, while the second branch is taken if the character is a `&`, `%`, `#`, a blankspace, or any simple one-byte character.

```
\newcommand\@retokenizechar[2]{%
  \isnextbyte[q]{\EscapeChar}{#1}%
  \if T\theresult%
  \edef\@ESCcode{\expandafter\@gobble#1}%
    \if\PipeCode\@ESCcode%
    \expandafter\def\csname#2\endcsname{\Pipe}\else
     \if\DollarCode\@ESCcode%
     \expandafter\def\csname#2\endcsname{\$}\else
      \if\CaratCode\@ESCcode%
      \expandafter\def\csname#2\endcsname{\Carat}\else
       \if\CircumflexCode\@ESCcode%
       \expandafter\def\csname#2\endcsname{\^}\else
        \if\TildeCode\@ESCcode%
        \expandafter\def\csname#2\endcsname{\~}\else
         \if\UmlautCode\@ESCcode%
         \expandafter\def\csname#2\endcsname{\"}\else
          \if\GraveCode\@ESCcode%
          \expandafter\def\csname#2\endcsname{\‘}\else
           \if\AcuteCode\@ESCcode%
           \expandafter\def\csname#2\endcsname{\’}\else
            \if\MacronCode\@ESCcode%
            \expandafter\def\csname#2\endcsname{\=}\else
             \if\OverdotCode\@ESCcode%
             \expandafter\def\csname#2\endcsname{\.}\else
              \if\LeftBraceCode\@ESCcode%
              \expandafter\def\csname#2\endcsname{\{}\else
               \if\RightBraceCode\@ESCcode%
               \expandafter\def\csname#2\endcsname{\}}\else
                \if\UnderscoreCode\@ESCcode%
                \expandafter\def\csname#2\endcsname{\_}\else
                 \if\DaggerCode\@ESCcode%
                 \expandafter\def\csname#2\endcsname{\dag}\else
                  \if\DoubleDaggerCode\@ESCcode%
                  \expandafter\def\csname#2\endcsname{\ddag}\else
                   \if\SectionSymbolCode\@ESCcode%
                   \expandafter\def\csname#2\endcsname{\S}\else
                    \if\PilcrowCode\@ESCcode%
                    \expandafter\def\csname#2\endcsname{\P}\else
```

```
                    \if\LBCode\@ESCcode%
                    \expandafter\def\csname#2\endcsname{\SaveLB}\else
                     \if\RBCode\@ESCcode%
                     \expandafter\def\csname#2\endcsname{\SaveRB}\else
\if\BreveCode\@ESCcode\expandafter\def\csname#2\endcsname{\u}\else
 \if\CaronCode\@ESCcode\expandafter\def\csname#2\endcsname{\v}\else
  \if\DoubleAcuteCode\@ESCcode\expandafter\def\csname#2\endcsname{\H}\else
   \if\CedillaCode\@ESCcode\expandafter\def\csname#2\endcsname{\c}\else
   \if\UnderdotCode\@ESCcode\expandafter\def\csname#2\endcsname{\d}\else
  \if\ArchJoinCode\@ESCcode\expandafter\def\csname#2\endcsname{\t}\else
 \if\LineUnderCode\@ESCcode\expandafter\def\csname#2\endcsname{\b}\else
\if\CopyrightCode\@ESCcode\expandafter\def\csname#2\endcsname{\copyright}\else
 \if\PoundsCode\@ESCcode\expandafter\def\csname#2\endcsname{\pounds}\else
  \if\AEscCode\@ESCcode\expandafter\def\csname#2\endcsname{\AE}\else
   \if\aescCode\@ESCcode\expandafter\def\csname#2\endcsname{\ae}\else
    \if\OEthelCode\@ESCcode\expandafter\def\csname#2\endcsname{\OE}\else
     \if\oethelCode\@ESCcode\expandafter\def\csname#2\endcsname{\oe}\else
      \if\AngstromCode\@ESCcode\expandafter\def\csname#2\endcsname{\AA}\else
       \if\angstromCode\@ESCcode\expandafter\def\csname#2\endcsname{\aa}\else
      \if\SlashedOCode\@ESCcode\expandafter\def\csname#2\endcsname{\O}\else
     \if\SlashedoCode\@ESCcode\expandafter\def\csname#2\endcsname{\o}\else
    \if\BarredlCode\@ESCcode\expandafter\def\csname#2\endcsname{\l}\else
   \if\BarredLCode\@ESCcode\expandafter\def\csname#2\endcsname{\L}\else
  \if\EszettCode\@ESCcode\expandafter\def\csname#2\endcsname{\ss}\else
 \expandafter\def\csname#2\endcsname{\undecipherable}%
  \fi
   \fi
    \fi
     \fi
      \fi
       \fi
      \fi
     \fi
    \fi
   \fi
  \fi
 \fi
\fi
 \fi
  \fi
   \fi
   \fi
  \fi
 \fi
\fi
                     \fi
                    \fi
                   \fi
                  \fi
                 \fi
```

```
                \fi
               \fi
              \fi
             \fi
            \fi
           \fi
          \fi
         \fi
        \fi
       \fi
      \fi
     \fi
    \fi
  \else
    \expandafter\ifx\expandafter\&#1%
                          \expandafter\def\csname#2\endcsname{\&}\else
      \expandafter\ifx\expandafter\%#1%
                          \expandafter\def\csname#2\endcsname{\%}\else
        \expandafter\ifx\expandafter\##1%
                          \expandafter\def\csname#2\endcsname{\#}\else
          \if\EncodedBlankSpace#1\expandafter\def\csname#2\endcsname{~}\else
            \expandafter\edef\csname#2\endcsname{#1}%
          \fi
        \fi
      \fi
    \fi
  \fi
}
```

`\@defineactions` This routine defines how encoded characters are to be treated by the `\ESCrotate` routine, depending on the [encoding, capitalization, blank, smbol, *etc.*] treatments that have been *a priori* specified.

```
% \@blankaction AND OTHER ...action'S ARE SET, DEPENDING ON VALUES OF
% TREATMENT FLAGS.  CHARS ARE EITHER ENCODED, DECODED, OR REMOVED.
\newcommand\@defineactions{%
% SET UP TREATMENT FOR SPACES, ENCODED SPACES, AND [REENCODED] SYMBOLS
  \if e\EncodingTreatment%
% ENCODE SPACES, KEEP ENCODED SPACES ENCODED, ENCODE SYMBOLS.
    \edef\@blankaction{\EncodedBlankSpace}%
    \def\@dollaraction{\EncodedDollar}%
    \def\@pipeaction{\EncodedPipe}%
    \def\@carataction{\EncodedCarat}%
    \def\@circumflexaction{\EncodedCircumflex}%
    \def\@tildeaction{\EncodedTilde}%
    \def\@umlautaction{\EncodedUmlaut}%
    \def\@graveaction{\EncodedGrave}%
    \def\@acuteaction{\EncodedAcute}%
    \def\@macronaction{\EncodedMacron}%
```

```
    \def\@overdotaction{\EncodedOverdot}%
    \def\@breveaction{\EncodedBreve}%
    \def\@caronaction{\EncodedCaron}%
    \def\@doubleacuteaction{\EncodedDoubleAcute}%
    \def\@cedillaaction{\EncodedCedilla}%
    \def\@underdotaction{\EncodedUnderdot}%
    \def\@archjoinaction{\EncodedArchJoin}%
    \def\@lineunderaction{\EncodedLineUnder}%
    \def\@copyrightaction{\EncodedCopyright}%
    \def\@poundsaction{\EncodedPounds}%
    \def\@leftbraceaction{\EncodedLeftBrace}%
    \def\@rightbraceaction{\EncodedRightBrace}%
    \def\@underscoreaction{\EncodedUnderscore}%
    \def\@daggeraction{\EncodedDagger}%
    \def\@doubledaggeraction{\EncodedDoubleDagger}%
    \def\@sectionsymbolaction{\EncodedSectionSymbol}%
    \def\@pilcrowaction{\EncodedPilcrow}%
    \def\@eszettaction{\EncodedEszett}%
    \def\@lbaction{\EncodedLB}%
    \def\@rbaction{\EncodedRB}%
    \if 2\AlphaCapsTreatment%
      \def\@AEscaction{\Encodedaesc}%
      \def\@OEthelaction{\Encodedoethel}%
      \def\@Angstromaction{\Encodedangstrom}%
      \def\@slashedOaction{\EncodedSlashedo}%
      \def\@barredLaction{\EncodedBarredl}%
    \else
      \def\@AEscaction{\EncodedAEsc}%
      \def\@OEthelaction{\EncodedOEthel}%
      \def\@Angstromaction{\EncodedAngstrom}%
      \def\@slashedOaction{\EncodedSlashedO}%
      \def\@barredLaction{\EncodedBarredL}%
    \fi
    \if 2\AlphaTreatment%
      \def\@aescaction{\EncodedAEsc}%
      \def\@oethelaction{\EncodedOEthel}%
      \def\@angstromaction{\EncodedAngstrom}%
      \def\@slashedoaction{\EncodedSlashedO}%
      \def\@barredlaction{\EncodedBarredL}%
    \else
      \def\@aescaction{\Encodedaesc}%
      \def\@oethelaction{\Encodedoethel}%
      \def\@angstromaction{\Encodedangstrom}%
      \def\@slashedoaction{\EncodedSlashedo}%
      \def\@barredlaction{\EncodedBarredl}%
    \fi
  \else
% EncodingTreatment=v or q:
% LEAVE SPACES ALONE; RESTORE ENCODED SPACES AND SYMBOLS
    \def\@blankaction{\BlankSpace}%
```

```
    \def\@dollaraction{\Dollar}%
    \def\@pipeaction{\Pipe}%
    \def\@carataction{\Carat}%
    \def\@circumflexaction{\Circumflex}%
    \def\@tildeaction{\Tilde}%
    \def\@umlautaction{\Umlaut}%
    \def\@graveaction{\Grave}%
    \def\@acuteaction{\Acute}%
    \def\@macronaction{\Macron}%
    \def\@overdotaction{\Overdot}%
    \def\@breveaction{\Breve}%
    \def\@caronaction{\Caron}%
    \def\@doubleacuteaction{\DoubleAcute}%
    \def\@cedillaaction{\Cedilla}%
    \def\@underdotaction{\Underdot}%
    \def\@archjoinaction{\ArchJoin}%
    \def\@lineunderaction{\LineUnder}%
    \def\@copyrightaction{\Copyright}%
    \def\@poundsaction{\Pounds}%
    \def\@leftbraceaction{\LeftBrace}%
    \def\@rightbraceaction{\RightBrace}%
    \def\@underscoreaction{\Underscore}%
    \def\@daggeraction{\Dagger}%
    \def\@doubledaggeraction{\DoubleDagger}%
    \def\@sectionsymbolaction{\SectionSymbol}%
    \def\@pilcrowaction{\Pilcrow}%
    \def\@eszettaction{\Eszett}%
    \def\@lbaction{\UnencodedLB}%
    \def\@rbaction{\UnencodedRB}%
    \if 2\AlphaCapsTreatment%
      \def\@AEscaction{\aesc}%
      \def\@OEthelaction{\oethel}%
      \def\@Angstromaction{\angstrom}%
      \def\@slashedOaction{\Slashedo}%
      \def\@barredLaction{\Barredl}%
    \else
      \def\@AEscaction{\AEsc}%
      \def\@OEthelaction{\OEthel}%
      \def\@Angstromaction{\Angstrom}%
      \def\@slashedOaction{\SlashedO}%
      \def\@barredLaction{\BarredL}%
    \fi
    \if 2\AlphaTreatment%
      \def\@aescaction{\AEsc}%
      \def\@oethelaction{\OEthel}%
      \def\@angstromaction{\Angstrom}%
      \def\@slashedoaction{\SlashedO}%
      \def\@barredlaction{\BarredL}%
    \else
      \def\@aescaction{\aesc}%
```

```
      \def\@oethelaction{\oethel}%
      \def\@angstromaction{\angstrom}%
      \def\@slashedoaction{\Slashedo}%
      \def\@barredlaction{\Barredl}%
    \fi
  \fi
% REMOVE SPACES AND ENCODED SPACES?
  \if 0\BlankTreatment%
    \edef\@blankaction{}%
  \fi
% REMOVE ENCODED SYMBOLS?
  \if 0\SymbolTreatment%
    \def\@dollaraction{}%
    \def\@pipeaction{}%
    \def\@carataction{}%
    \def\@circumflexaction{}%
    \def\@tildeaction{}%
    \def\@umlautaction{}%
    \def\@graveaction{}%
    \def\@acuteaction{}%
    \def\@macronaction{}%
    \def\@overdotaction{}%
    \def\@breveaction{}%
    \def\@caronaction{}%
    \def\@doubleacuteaction{}%
    \def\@cedillaaction{}%
    \def\@underdotaction{}%
    \def\@archjoinaction{}%
    \def\@lineunderaction{}%
    \def\@copyrightaction{}%
    \def\@poundsaction{}%
    \def\@leftbraceaction{}%
    \def\@rightbraceaction{}%
    \def\@underscoreaction{}%
    \def\@daggeraction{}%
    \def\@doubledaggeraction{}%
    \def\@sectionsymbolaction{}%
    \def\@pilcrowaction{}%
    \def\@lbaction{}%
    \def\@rbaction{}%
  \fi
% REMOVE ENCODED ALPHACAPS?
  \if 0\AlphaCapsTreatment%
    \def\@AEscaction{}%
    \def\@OEthelaction{}%
    \def\@Angstromaction{}%
    \def\@slashedOaction{}%
    \def\@barredLaction{}%
  \fi
% REMOVE ENCODED ALPHA?
```

```
  \if 0\AlphaTreatment%
    \def\@aescaction{}%
    \def\@oethelaction{}%
    \def\@angstromaction{}%
    \def\@slashedoaction{}%
    \def\@barredlaction{}%
    \def\@eszettaction{}%
  \fi
}
```

`\@forcecapson` Force capitalization of strings processed by `\substring` for the time being.

```
\newcommand\@forcecapson{%
  \def\AlphaTreatment{2}%
  \def\AlphaCapsTreatment{1}%
}
```

`\@relaxcapson` Restore prior treatments following a period of enforced capitalization.

```
\newcommand\@relaxcapson{%
  \let\AlphaTreatment\SaveAlphaTreatment%
  \let\AlphaCapsTreatment\SaveAlphaCapsTreatment%
  \@defineactions%
}
```

`\@decodepointer` As pertains to arguments 3 and 4 of `\substring`, this routine implements use of the `$` character to mean *end-of-string*, and `$-{`*number*`}` for addressing relative to the *end-of-string*.

```
\newcommand\@decodepointer[2][\value{@stringsize}]{%
  \isnextbyte[q]{$}{#2}%
  \if T\theresult%
    \isnextbyte[q]{-}{\expandafter\@gobble#2}%
    \if T\theresult%
      \setcounter{@@@letterindex}{#1}%
      \@gobblearg{#2}{2}%
      \addtocounter{@@@letterindex}{-\gobbledword}%
      \edef\fromtoindex{\value{@@@letterindex}}%
    \else
      \edef\fromtoindex{#1}%
    \fi
  \else
    \edef\fromtoindex{#2}%
  \fi
}
```

`\@getstringlength` Get's string length of `#1`, puts result in counter `#2`.

```
\newcommand\@getstringlength[2]{%
  \edef\@@teststring{#1\endofstring}%
```

```
  \ifthenelse{\equal{\@@teststring}{\endofstring}}%
  {\setcounter{#2}{0}}%
  {%
    \setcounter{@gobblesize}{1}%
    \whiledo{\value{@gobblesize} < \@MAXSTRINGSIZE}{%
%
      \@gobblearg{\@@teststring}{1}%
      \edef\@@teststring{\gobbledword}%
      \ifthenelse{\equal{\@@teststring}{\endofstring}}%
        {\setcounter{#2}{\value{@gobblesize}}%
          \setcounter{@gobblesize}{\@MAXSTRINGSIZE}}%
        {\addtocounter{@gobblesize}{1}}%
    }%
  }%
}
```

`\@gobblearg` Gobble first `#2` characters from string `#1`. The result is stored in `\gobbledword`. Two-byte escape sequences, when encountered, count as a single gobble.

```
\newcommand\@gobblearg[2]{%
  \setcounter{@letterindex}{0}%
  \setcounter{@gobbleindex}{#2}%
  \edef\gobbledword{#1}%
  \whiledo{\value{@letterindex} < \value{@gobbleindex}}{%
    \isnextbyte[q]{\EscapeChar}{\gobbledword}%
    \if T\theresult%
%     GOBBLE ESCAPE CHARACTER
      \edef\gobbledword{\@DiscardNextChar{\gobbledword}}%
    \fi
%   GOBBLE NORMAL CHARACTER OR ESCAPE CODE
    \edef\gobbledword{\@DiscardNextChar{\gobbledword}}%
    \addtocounter{@letterindex}{1}%
  }%
}
```

`\@DiscardNextChar` Remove the next character from the argument string. Since `\@gobble` skips spaces, the routine must first look for the case of a leading blankspace. If none is found, proceed with a normal `\@gobble`. Note: as per LaTeX convention, `\@DiscardNextChar` treats double/multi-softspaces as single space.

```
\newcommand\@DiscardNextChar[1]{%
  \expandafter\if\expandafter\BlankSpace#1\else
    \expandafter\@gobble#1%
  \fi
}
```

`\@convertsymboltostring` Routine for converting an encodable symbol (`#3`) into string (`#4`), for every occurance in the given string `#2`.

```
\newcommand\@convertsymboltostring[4][v]{%
```

```
  \def\@fromcode{#3}%
  \def\@tostring{#4}%
  \def\EncodingTreatment{e}%
  \substring[e]{#2}{1}{\@MAXSTRINGSIZE}%
  \@convertoff%
  \if e#1\else\substring[#1]{\thestring}{1}{\@MAXSTRINGSIZE}\fi%
}
```

`\@convertbytetostring` Routine for converting an plain byte (`#3`) into string (`#4`), for every occurance in the given string `#2`.

```
\newcommand\@convertbytetostring[4][v]{%
  \def\@frombyte{#3}%
  \def\@tostring{#4}%
  \def\EncodingTreatment{e}%
  \substring[e]{#2}{1}{\@MAXSTRINGSIZE}%
  \@convertoff%
  \if e#1\else\substring[#1]{\thestring}{1}{\@MAXSTRINGSIZE}\fi%
}
```

`\@treatleadingspaces` This routine will address the leading spaces of string `#2`. If argument `#3` is an 'x' character, those leading spaces will be deleted from the string. Otherwise, those leading spaces will be rotated to the end of the string.

```
\newcommand\@treatleadingspaces[3][v]{\+%
  \defaultTreatments%
  \edef\thestring{#2}%
  \@getstringlength{\thestring}{@stringsize}%
  \setcounter{@maxrotation}{\value{@stringsize}}%
  \setcounter{@letterindex}{0}%
  \whiledo{\value{@letterindex} < \value{@maxrotation}}{%
    \addtocounter{@letterindex}{1}%
    \isnextbyte[q]{\EncodedBlankSpace}{\thestring}%
    \if F\theresult\isnextbyte[q]{\BlankSpace}{\thestring}\fi%
    \if T\theresult%
      \isnextbyte[q]{#3}{x}%
      \if F\theresult%
%       NORMAL OR ENCODED BLANK... ROTATE IT
        \edef\thestring{\rotate{\thestring}}%
      \else
%       NORMAL OR ENCODED BLANK... DELETE IT (IF 3rd ARG=X)
        \@gobblearg{\thestring}{1}%
        \edef\thestring{\gobbledword}%
      \fi
    \else
      \setcounter{@maxrotation}{\value{@letterindex}}%
    \fi
  }\?%
  \substring[#1]{\thestring}{1}{\@MAXSTRINGSIZE}%
}
```

`\@convertoff` This routine is an initialization routine to guarantee that there is no conversion of `\@frombyte` to `\@tostring`, until further notice. It accomplishes this by setting up such that subsequent `\if\@frombyte` and `\if\@fromcode` clauses will automatically fail.

```
\newcommand\@convertoff{\def\@frombyte{xy}\def\@tostring{}%
                        \def\@fromcode{xy}}
\@convertoff
```

`\@rotate` The following code is the engine of the string manipulation routine. It is a tree of successive LaTeX commands (each of which is composed of an `\if...` cascade) which have the net effect of rotating the first letter of the string into the last position. Depending on modes set by `\@defineactions` and `\defaultTreatments`, various characters are either encoded, decoded, or removed. Note: `\@rotate` loses track of double/multi-spaces, per LaTeX convention, unless encoded blanks (`~`) are used.

```
\newcommand\@rotate[1]{%
% CHECK BYTE CONVERSION TEST FIRST
 \if \@frombyte#1\@tostring\else
% MUST CHECK FOR MULTI-BYTE CHARACTERS NEXT, SO THAT ENCODING CHARACTER
% ISN'T MISTAKEN FOR A NORMAL CHARACTER LATER IN MACRO.
  \if 0\SymbolTreatment%
    \@removeExpandableSymbols{#1}%
  \else
    \@rotateExpandableSymbols{#1}%
  \fi
 \fi
}

\newcommand\@rotateExpandableSymbols[1]{%
% INDIRECT (EXPANDABLE) SYMBOLS
  \expandafter\ifx\expandafter\&#1\&\else
   \expandafter\ifx\expandafter\%#1\%\else
    \expandafter\ifx\expandafter\##1\#\else
     \@rotateBlankSpaces{#1}%
    \fi
   \fi
  \fi
}

\newcommand\@removeExpandableSymbols[1]{%
% INDIRECT (EXPANDABLE) SYMBOLS
  \expandafter\ifx\expandafter\&#1\else
   \expandafter\ifx\expandafter\%#1\else
    \expandafter\ifx\expandafter\##1\else
     \@rotateBlankSpaces{#1}%
    \fi
   \fi
  \fi
```

```
}

\newcommand\@rotateBlankSpaces[1]{%
  \expandafter\ifx\expandafter$#1$\else% <---RETAIN GOING INTO/FROM MATH MODE
% THE FOLLOWING FINDS TILDES, BUT MUST COME AFTER EXPANDABLE SYMBOL
% SEARCH, OR ELSE IT FINDS THEM TOO, BY MISTAKE.
   \if \EncodedBlankSpace#1\@blankaction\else% <--- FINDS REENCODED TILDE
%   THE FOLLOWING SHOULD FIND TILDES, BUT DOESN'T... THUS, COMMENTED OUT.
%   \expandafter\ifx\expandafter\EncodedBlankSpace#1\@blankaction\else
     \if \BlankSpace#1\@blankaction\else
      \if 2\AlphaTreatment%
        \@chcaseAlpha{#1}%
      \else
        \if 0\AlphaTreatment%
          \@removeAlpha{#1}%
        \else
          \@rotateAlpha{#1}%
        \fi
      \fi
     \fi
%   \fi
   \fi
  \fi
}

\newcommand\@rotateAlpha[1]{%
% LOWERCASE
  \if a#1a\else
   \if b#1b\else
    \if c#1c\else
     \if d#1d\else
      \if e#1e\else
       \if f#1f\else
       \if g#1g\else
        \if h#1h\else
         \if i#1i\else
          \if j#1j\else
           \if k#1k\else
            \if l#1l\else
             \if m#1m\else
              \if n#1n\else
               \if o#1o\else
                \if p#1p\else
                 \if q#1q\else
                  \if r#1r\else
                   \if s#1s\else
                    \if t#1t\else
                     \if u#1u\else
                      \if v#1v\else
                       \if w#1w\else
```

```
                        \if x#1x\else
                         \if y#1y\else
                          \if z#1z\else
                           \if 2\AlphaCapsTreatment%
                             \@chcaseAlphaCaps{#1}%
                           \else
                             \if 0\AlphaCapsTreatment%
                               \@removeAlphaCaps{#1}%
                             \else
                               \@rotateAlphaCaps{#1}%
                             \fi
                           \fi
                          \fi
                         \fi
                        \fi
                       \fi
                      \fi
                     \fi
                    \fi
                   \fi
                  \fi
                 \fi
                \fi
               \fi
              \fi
             \fi
            \fi
           \fi
          \fi
         \fi
        \fi
       \fi
      \fi
     \fi
    \fi
   \fi
  \fi
}

\newcommand\@removeAlpha[1]{%
% LOWERCASE
  \if a#1\else
   \if b#1\else
    \if c#1\else
     \if d#1\else
      \if e#1\else
       \if f#1\else
        \if g#1\else
         \if h#1\else
```

```
      \if i#1\else
       \if j#1\else
        \if k#1\else
         \if l#1\else
          \if m#1\else
           \if n#1\else
            \if o#1\else
             \if p#1\else
              \if q#1\else
               \if r#1\else
                \if s#1\else
                 \if t#1\else
                  \if u#1\else
                   \if v#1\else
                    \if w#1\else
                     \if x#1\else
                      \if y#1\else
                       \if z#1\else
                        \if 2\AlphaCapsTreatment%
                          \@chcaseAlphaCaps{#1}%
                        \else
                          \if 0\AlphaCapsTreatment%
                            \@removeAlphaCaps{#1}%
                          \else
                            \@rotateAlphaCaps{#1}%
                          \fi
                        \fi
                       \fi
                      \fi
                     \fi
                    \fi
                   \fi
                  \fi
                 \fi
                \fi
               \fi
              \fi
             \fi
            \fi
           \fi
          \fi
         \fi
        \fi
       \fi
      \fi
     \fi
    \fi
```

```
    \fi
   \fi
  \fi
}

\newcommand\@chcaseAlpha[1]{%
% LOWERCASE TO UPPERCASE
  \if a#1A\else
   \if b#1B\else
    \if c#1C\else
     \if d#1D\else
      \if e#1E\else
       \if f#1F\else
        \if g#1G\else
         \if h#1H\else
          \if i#1I\else
           \if j#1J\else
            \if k#1K\else
             \if l#1L\else
              \if m#1M\else
               \if n#1N\else
                \if o#1O\else
                 \if p#1P\else
                  \if q#1Q\else
                   \if r#1R\else
                    \if s#1S\else
                     \if t#1T\else
                      \if u#1U\else
                       \if v#1V\else
                        \if w#1W\else
                         \if x#1X\else
                          \if y#1Y\else
                           \if z#1Z\else
                            \if 2\AlphaCapsTreatment%
                              \@chcaseAlphaCaps{#1}%
                            \else
                              \if 0\AlphaCapsTreatment%
                                \@removeAlphaCaps{#1}%
                              \else
                                \@rotateAlphaCaps{#1}%
                              \fi
                            \fi
                           \fi
                          \fi
                         \fi
                        \fi
                       \fi
                      \fi
                     \fi
                    \fi
                   \fi
                  \fi
                 \fi
                \fi
               \fi
              \fi
             \fi
            \fi
           \fi
          \fi
         \fi
        \fi
       \fi
      \fi
     \fi
    \fi
   \fi
```

```
                  \fi
                 \fi
                \fi
               \fi
              \fi
             \fi
            \fi
           \fi
          \fi
         \fi
        \fi
       \fi
      \fi
     \fi
    \fi
  \fi
}

\newcommand\@rotateAlphaCaps[1]{%
% UPPERCASE
  \if A#1A\else
   \if B#1B\else
    \if C#1C\else
     \if D#1D\else
      \if E#1E\else
       \if F#1F\else
        \if G#1G\else
         \if H#1H\else
          \if I#1I\else
           \if J#1J\else
            \if K#1K\else
             \if L#1L\else
              \if M#1M\else
               \if N#1N\else
                \if O#1O\else
                 \if P#1P\else
                  \if Q#1Q\else
                   \if R#1R\else
                    \if S#1S\else
                     \if T#1T\else
                      \if U#1U\else
                       \if V#1V\else
                        \if W#1W\else
                         \if X#1X\else
                          \if Y#1Y\else
                           \if Z#1Z\else
                            \if 0\NumeralTreatment%
                              \@removeNumerals{#1}%
```

```
                            \else
                              \@rotateNumerals{#1}%
                            \fi
                           \fi
                          \fi
                         \fi
                        \fi
                       \fi
                      \fi
                     \fi
                    \fi
                   \fi
                  \fi
                 \fi
                \fi
               \fi
              \fi
             \fi
            \fi
           \fi
          \fi
         \fi
        \fi
       \fi
      \fi
     \fi
    \fi
   \fi
  \fi
}

\newcommand\@removeAlphaCaps[1]{%
% UPPERCASE
  \if A#1\else
   \if B#1\else
    \if C#1\else
     \if D#1\else
      \if E#1\else
       \if F#1\else
        \if G#1\else
         \if H#1\else
          \if I#1\else
           \if J#1\else
            \if K#1\else
             \if L#1\else
              \if M#1\else
               \if N#1\else
                \if O#1\else
                 \if P#1\else
                  \if Q#1\else
```

```
                    \if R#1\else
                     \if S#1\else
                      \if T#1\else
                       \if U#1\else
                        \if V#1\else
                         \if W#1\else
                          \if X#1\else
                           \if Y#1\else
                            \if Z#1\else
                             \if 0\NumeralTreatment%
                               \@removeNumerals{#1}%
                             \else
                               \@rotateNumerals{#1}%
                             \fi
                            \fi
                           \fi
                          \fi
                         \fi
                        \fi
                       \fi
                      \fi
                     \fi
                    \fi
                   \fi
                  \fi
                 \fi
                \fi
               \fi
              \fi
             \fi
            \fi
           \fi
          \fi
         \fi
        \fi
       \fi
      \fi
     \fi
    \fi
   \fi
  \fi
}

\newcommand\@chcaseAlphaCaps[1]{%
% UPPERCASE TO LOWERCASE
  \if A#1a\else
   \if B#1b\else
    \if C#1c\else
     \if D#1d\else
      \if E#1e\else
       \if F#1f\else
```

```
      \if G#1g\else
       \if H#1h\else
        \if I#1i\else
         \if J#1j\else
          \if K#1k\else
           \if L#1l\else
            \if M#1m\else
             \if N#1n\else
              \if O#1o\else
               \if P#1p\else
                \if Q#1q\else
                 \if R#1r\else
                  \if S#1s\else
                   \if T#1t\else
                    \if U#1u\else
                     \if V#1v\else
                      \if W#1w\else
                       \if X#1x\else
                        \if Y#1y\else
                         \if Z#1z\else
                          \if 0\NumeralTreatment%
                            \@removeNumerals{#1}%
                          \else
                            \@rotateNumerals{#1}%
                          \fi
                         \fi
                        \fi
                       \fi
                      \fi
                     \fi
                    \fi
                   \fi
                  \fi
                 \fi
                \fi
               \fi
              \fi
             \fi
            \fi
           \fi
          \fi
         \fi
        \fi
       \fi
      \fi
     \fi
    \fi
   \fi
  \fi
```

```
  \fi
}

\newcommand\@rotateNumerals[1]{%
% NUMERALS
  \if 1#11\else
   \if 2#12\else
    \if 3#13\else
     \if 4#14\else
      \if 5#15\else
       \if 6#16\else
        \if 7#17\else
         \if 8#18\else
          \if 9#19\else
           \if 0#10\else
            \if 0\PunctuationTreatment%
              \@removePunctuation{#1}%
            \else
              \@rotatePunctuation{#1}%
            \fi
           \fi
          \fi
         \fi
        \fi
       \fi
      \fi
     \fi
    \fi
   \fi
  \fi
}

\newcommand\@removeNumerals[1]{%
% NUMERALS
  \if 1#1\else
   \if 2#1\else
    \if 3#1\else
     \if 4#1\else
      \if 5#1\else
       \if 6#1\else
        \if 7#1\else
         \if 8#1\else
          \if 9#1\else
           \if 0#1\else
            \if 0\PunctuationTreatment%
              \@removePunctuation{#1}%
            \else
              \@rotatePunctuation{#1}%
            \fi
           \fi
```

```
          \fi
         \fi
        \fi
       \fi
      \fi
     \fi
    \fi
   \fi
  \fi
}

\newcommand\@rotatePunctuation[1]{%
% PUNCTUATION
  \if ;#1;\else
   \if :#1:\else
    \if '#1'\else
     \if "#1"\else
      \if ,#1,\else
       \if .#1.\else
        \if ?#1?\else
         \if `#1`\else
          \if !#1!\else
           \if 0\SymbolTreatment%
             \@removeDirectSymbols{#1}%
           \else
             \@rotateDirectSymbols{#1}%
           \fi
          \fi
         \fi
        \fi
       \fi
      \fi
     \fi
    \fi
   \fi
  \fi
}

\newcommand\@removePunctuation[1]{%
% PUNCTUATION
  \if ;#1\else
   \if :#1\else
    \if '#1\else
     \if "#1\else
      \if ,#1\else
       \if .#1\else
        \if ?#1\else
         \if `#1\else
          \if !#1\else
           \if 0\SymbolTreatment%
```

```
            \@removeDirectSymbols{#1}%
           \else
            \@rotateDirectSymbols{#1}%
           \fi
          \fi
         \fi
        \fi
       \fi
      \fi
     \fi
    \fi
   \fi
  \fi
}

\newcommand\@rotateDirectSymbols[1]{%
% DIRECT SYMBOLS
  \if /#1/\else
   \if @#1@\else
    \if *#1*\else
     \if (#1(\else
      \if )#1)\else
       \if -#1-\else
        \if _#1_\else
         \if =#1=\else
          \if +#1+\else
           \if [#1[\else
            \if ]#1]\else
             \if ^#1^\else% <--FOR SUPERSCRIPTS, NOT \^
              \if <#1<\else
               \if >#1>\else
                \if |#1|\else
                 \@rotateUndecipherable{#1}%
                \fi
               \fi
              \fi
             \fi
            \fi
           \fi
          \fi
         \fi
        \fi
       \fi
      \fi
     \fi
    \fi
   \fi
  \fi
}
```

```
\newcommand\@removeDirectSymbols[1]{%
% DIRECT SYMBOLS
  \if /#1\else
   \if @#1\else
    \if *#1\else
     \if (#1\else
      \if )#1\else
       \if -#1\else
        \if _#1\else
         \if =#1\else
          \if +#1\else
           \if [#1\else
            \if ]#1\else
             \if ^#1\else% <--FOR SUPERSCRIPTS, NOT \^
              \if <#1\else
               \if >#1\else
                \if |#1\else
                 \@rotateUndecipherable{#1}%
                \fi
               \fi
              \fi
             \fi
            \fi
           \fi
          \fi
         \fi
        \fi
       \fi
      \fi
     \fi
    \fi
   \fi
  \fi
}

\newcommand\@rotateUndecipherable[1]{%
% REPLACE UNDECIPHERABLE SYMBOL WITH A TOKEN CHARACTER (DEFAULT .)
  \expandafter\@gobble#1\undecipherable%
% DONE... CLOSE UP SHOP
}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
⟨/package⟩
```