

The `stringstrings` Package

Extensive array of string manipulation routines for cosmetic and programming application

Steven B. Segletes
steven@arl.army.mil

2009/10/01
v1.04

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	2
2	Philosophy of Operation	4
3	Configuration Commands	6
4	Commands to Manipulate Strings	7
5	Commands to Extract String Information	11
6	Commands to Test Strings	12
7	Disclaimers	14

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 beginning 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 L^AT_EX symbols (*e.g.*, `\$, \{, \}, \AE, \oe`, etc.) which expand to more than a single byte. If I was more savvy on L^AT_EX 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 L^AT_EX symbols at the last moment, as the finalized strings were handed back to the user.

There are also several tokens, like `\dag, \ddag, \P, \d, \t, \b`, and `\copyright` 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 occurrence of L^AT_EX 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 L^AT_EX 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 command 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 through 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 inadvertently overwrite) the same sets of variables used by other routines. Furthermore, there is that 'ol left-to-right philosophy of L^AT_EX to contend with.

Instead, for the case of commands that manipulate strings, the expanded `\thestring` (*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.

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.

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`.

In addition to providing tokenized strings that can be passed to other \LaTeX packages, `\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 (see Table 1 for summary).

Table 1: **Execution Modes of `stringstrings` Commands**

Mode	Coding	Use when result is	<code>\thestring</code> is
<code>[v]</code> verbose	decoded or retokenized	final	echoed
<code>[q]</code> quiet	decoded or retokenized	final	not echoed
<code>[e]</code> encoded	encoded	intermediate	not echoed

Moving on to commands that extract string information, this class of commands (unless otherwise noted) output their result into a token which is given the name `\theresult`. This token does not contain 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`, most of these commands use the `\testcondition{string}` `\ifcondition` 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, Å, Ø, and Ł). 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, ø, ł, 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 and diacritical marks: / * () - = + [] < > & \& \% \# \{ \} _ \\$ † ‡ § ¶ £ ¤ © ã â Ã Ä Å å æ ç è é ê ë ÿ x̂ x̃ x̄ x̅ x̆ ẋ ẍ x̉ x̊ x̋ x̌ x̍ x̎ x̏ x̐ x̑ x̒ x̓ x̔ x̕ x̖ x̗ x̘ x̙ x̚ x̛ x̜ x̝ x̞ x̟ x̠ x̡ x̢ x̣ x̤ x̥ x̦ x̧ x̨ x̩ x̪ x̫ x̬ x̭ x̮ x̯ x̰ x̱ x̲ x̳ x̴ x̵ x̶ x̷ x̸ x̹ x̺ x̀ x́ x͂ x̓ ẍ́ xͅ x͆ x͇ x͈ x͉ x͊ x͋ x͌ x͍ x͎ x͏ x͐ x͑ x͒ x͓ x͔ x͕ x͖ x͗ x͘ x͙ x͚ x͛ x͜ x͝ x͞ x͟ x͠ x͡ x͢ xͣ xͤ xͥ xͦ xͧ xͨ xͩ xͪ xͫ xͬ xͭ xͮ xͯ as well as @, math and text carats, 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 applies 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`, `\oe`, `\~`, `\'`, `\,`, `\=`, `\.`, `\u`, `\v`, `\H`, `\c`, `\d`, `\t`, `\b`, `\copyright`, `\pounds`, `\L`, `\l`, and `\ss`. In addition, pipes, text carats, and hard spaces (`~`) are encoded as well. The command `\?` restores the standard L^AT_EX 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}`
- `\change[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].
- \change-case** The command **\change-case** 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 occurrences of *from-char* in the original string are replaced. 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 occurrences 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 L^AT_EX 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 L^AT_EX 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`, `\d`, `\t`, `\b`, `\copyright`, 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`.

<code>\stringlength</code>	The command <code>\stringlength</code> returns the length of <i>string</i> in characters (not bytes). Default mode is <code>[v]</code> .
<code>\findchars</code>	The command <code>\findchars</code> checks to see if the character <i>match-char</i> occurs anywhere in <i>string</i> . The number of occurrences is stored in <code>\theresult</code> and, if in verbose mode, printed. If it is desired to find blankspaces, <i>match-char</i> should be set to <code>{~}</code> and not <code>{ }</code> . Default mode is <code>[v]</code> .
<code>\findwords</code>	The command <code>\findwords</code> checks to see if the string <i>match-string</i> occurs anywhere in <i>string</i> . The number of occurrences is stored in <code>\theresult</code> and, if in verbose mode, printed. If it is desired to find blankspaces, those characters in <i>match-string</i> should be set to hardspaces (<i>i.e.</i> , tildes) and not softspaces (<i>i.e.</i> , blanks), regardless of how they are defined in <i>string</i> . Default mode is <code>[v]</code> .
<code>\whereischar</code>	The command <code>\whereischar</code> checks to see where the character <i>match-char</i> first occurs in <i>string</i> . The location of that occurrence is stored in <code>\theresult</code> and, if in verbose mode, printed. If the character is not found, <code>\theresult</code> is set to a value of 0. If it is desired to find blankspaces, <i>match-char</i> should be set to <code>{~}</code> and not <code>{ }</code> . Default mode is <code>[v]</code> .
<code>\whereisword</code>	The command <code>\whereisword</code> checks to see where the string <i>match-string</i> first occurs in <i>string</i> . The location of that occurrence is stored in <code>\theresult</code> and, if in verbose mode, printed. If <i>match-string</i> is not found, <code>\theresult</code> is set to a value of 0. If it is desired to find blankspaces, those characters in <i>match-string</i> should be set to hardspaces (<i>i.e.</i> , tildes) and not softspaces (<i>i.e.</i> , blanks), regardless of how they are defined in <i>string</i> . Default mode is <code>[v]</code> .
<code>\wordcount</code>	The command <code>\wordcount</code> counts the number of space-separated words that occur in <i>string</i> . Default mode is <code>[v]</code> .
<code>\getargs</code>	The command <code>\getargs</code> 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 `\narg` 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]`; `[e]` mode is permitted, while `[v]` mode is disabled.

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 the style 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 of diacritical marks, which count as separate symbols from the characters they mark. For example, `\^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` (`\Edipus`) 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.

One quirk worthy of particular note concerns the tabbing character, meaning `&` as opposed to `\&` (which is handled without problem). As of version 1.01, `stringstrings` has the capability to operate on arguments containing the ampersand `&`, normally reserved as the \LaTeX tabbing character. However, one adverse by-product is that `&` characters returned in `\thestring` lose their catcode-4 value, and thus lose their ability to function as tabbing characters. In the following example,

```
\caseupper[q]{a & b & c & d}
\begin{tabular}{|l|c|c|c|}
\hline
\thestring\\
\hline
\end{tabular}
```

will produce

A	&	B	&	C	&	D
---	---	---	---	---	---	---

 instead of the desired

A		B		C		D
---	--	---	--	---	--	---

.

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 2 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 three 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 L^AT_EX’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 L^AT_EX, 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 `\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`), `ˆ` (`\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 inadvertently 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,

Table 2: Problematic Characters/Tokens and stringstrings Solutions

L ^A T _E X	Symbol/Name	Problem/Solution
{ }	begin group end group	Cannot use { and } in <code>stringstrings</code> arguments. However, use <code>\LB...\RB</code> in lieu of <code>{...}</code> ; manipulate string in [e] mode & <code>\retokenize</code>
\dag \ddag \P \d \t \b \copyright	† Dagger ‡ Double Dagger ¶ Pilcrow x Underdot x̂ Joining Arch x Letter Underline © Copyright	Cannot <code>\edef</code> these tokens; Thus, [v] mode fails with both OT1 and T1 encoding; manipulate string in [e] mode & <code>\retokenize</code>
_ \{ \} \S \c \pounds	- Underscore { Left Curly Brace } Right Curly Brace § Section Symbol x̣ Cedilla £ Pounds	Cannot <code>\edef</code> with OT1 encoding; either <code>\renewcommand\encodingdefault{T1}</code> , or manipulate string in [e] mode & <code>\retokenize</code> . With OT1, <code>\S</code> , <code>\c</code> and <code>\pounds</code> break <code>stringstrings</code> [v] mode.
\	<code>stringstrings</code> Pipe Char. (T1) — (OT1)	Distinct from , the <code>stringstrings</code> encoded-escape character
\\$ \carat \^ \' \" \~ \' \. \= \u \v \H \ss \AE \ae \OE \oe \AA \aa \O \o \L \l ~	\$ Dollar ^ (text mode) x̂ Circumflex x́ Acute ẍ Umlaut x̃ Tilde x̀ Grave ẋ Overdot x̄ Macron x̆ Breve ẍ́ Caron ẍ́ Double Acute ß Eszett Æ æ æsc Œ œ œthel Å å angstrom Ø ø slashed O Ł ł barred L ~ Hardspace	Either cannot <code>\edef</code> , or cannot identify uniquely with <code>\if</code> construct, or expanded character is more than one byte. <i>However,</i> Use these characters freely, <code>stringstrings</code> encoding functions transparently with them. <code>\retokenize</code> also works
\$ ^ _	begin/end math mode math superscript math subscript	These characters pose no difficulties; However, cannot extract substring that breaks in middle of math mode. Other math mode symbols NOT supported.
&	ampersand	Version 1.01 <code>stringstrings</code> can manipulate the ampersand. However, returned strings containing the & character lose their catcode-4 status, making them unfit for use as tabbing characters.

`\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 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
4 \catcode'\&=12

ifthen This package makes wide use of the ifthen style package.

5 \usepackage{ifthen}

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

6 \def\@MAXSTRINGSIZE{500}

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

Save the symbols which will get redefined stringstrings encoding.

10 \let\SaveDollar\$
11 \let\SaveHardspace~
12 \let\SaveCircumflex^
13 \let\SaveTilde~
14 \let\SaveUmlaut\"
15 \let\SaveGrave\'

```

```

16 \let\SaveAcute\'
17 \let\SaveMacron\=
18 \let\SaveOverdot\.
19 \let\SaveBreve\u
20 \let\SaveCaron\v
21 \let\SaveDoubleAcute\H
22 \let\SaveCedilla\c
23 \let\SaveUnderdot\d
24 \let\SaveArchJoin\t
25 \let\SaveLineUnder\b
26 \let\SaveCopyright\copyright
27 \let\SavePounds\pounds
28 \let\SaveLeftBrace\{
29 \let\SaveRightBrace\}
30 \let\SaveUnderscore\_
31 \let\SaveDagger\dag
32 \let\SaveDoubleDagger\ddag
33 \let\SaveSectionSymbol\S
34 \let\SavePilcrow\P
35 \let\SaveAEsc\AE
36 \let\Saveaesc\ae
37 \let\SaveOEthel\OE
38 \let\Saveoeethel\oe
39 \let\SaveAngstrom\AA
40 \let\Saveangstrom\aa
41 \let\SaveSlashedO\O
42 \let\SaveSlashedo\o
43 \let\SaveBarredL\L
44 \let\SaveBarredl\l
45 \let\SaveEszett\ss
46 \let\SaveLB{
47 \let\SaveRB}

```

The BlankSpace character is the only character which is reencoded with a 1-byte re-encoding... in this case the OE character.

```

48 \def\EncodedBlankSpace{\SaveOEthel}
49 \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.

```

50 \def\EscapeChar{|}
51
52 % |O IS AN ENCODED |, ACCESSED VIA \|
53 \def\PipeCode{O}
54 \def\EncodedPipe{\EscapeChar\PipeCode}
55 \def\Pipe{|}
56 \let\|\EncodedPipe
57

```

```

58 % |1 IS AN ENCODED \$
59 \def\DollarCode{1}
60 \def\EncodedDollar{\EscapeChar\DollarCode}
61 % THE FOLLOWING IS NEEDED TO KEEP OT1 ENCODING FROM BREAKING;
62 % IT PROVIDES AN ADEQUATE BUT NOT IDEAL ENVIRONMENT FOR T1 ENCODING
63 \def\Dollar{\symbol{36}}
64 % THE FOLLOWING IS BETTER FOR T1 ENCODING, BUT BREAKS OT1 ENCODING
65 %\def\Dollar{\SaveDollar}
66
67 % |2 IS AN ENCODED ^ FOR USE IN TEXT MODE, ACCESSED VIA \carat
68 \def\CaratCode{2}
69 \def\EncodedCarat{\EscapeChar\CaratCode}
70 \def\Carat{\symbol{94}}
71 \let\carat\EncodedCarat
72
73 % |4 IS AN ENCODED \{
74 \def\LeftBraceCode{4}
75 \def\EncodedLeftBrace{\EscapeChar\LeftBraceCode}
76 % THE FOLLOWING IS NEEDED TO KEEP OT1 ENCODING FROM BREAKING;
77 % IT PROVIDES AN ADEQUATE BUT NOT IDEAL ENVIRONMENT FOR T1 ENCODING
78 \def\LeftBrace{\symbol{123}}
79 % THE FOLLOWING IS BETTER FOR T1 ENCODING, BUT BREAKS OT1 ENCODING
80 %\def\LeftBrace{\SaveLeftBrace}
81
82 % |5 IS AN ENCODED \}
83 \def\RightBraceCode{5}
84 \def\EncodedRightBrace{\EscapeChar\RightBraceCode}
85 % THE FOLLOWING IS NEEDED TO KEEP OT1 ENCODING FROM BREAKING;
86 % IT PROVIDES AN ADEQUATE BUT NOT IDEAL ENVIRONMENT FOR T1 ENCODING
87 \def\RightBrace{\symbol{125}}
88 % THE FOLLOWING IS BETTER FOR T1 ENCODING, BUT BREAKS OT1 ENCODING
89 %\def\RightBrace{\SaveRightBrace}
90
91 % |6 IS AN ENCODED \_
92 \def\UnderscoreCode{6}
93 \def\EncodedUnderscore{\EscapeChar\UnderscoreCode}
94 \def\Underscore{\symbol{95}}
95 %\def\Underscore{\SaveUnderscore}
96
97 % |7 IS AN ENCODED \^
98 \def\CircumflexCode{7}
99 \def\EncodedCircumflex{\EscapeChar\CircumflexCode}
100 \def\Circumflex{\noexpand\SaveCircumflex}
101
102 % |8 IS AN ENCODED \~
103 \def\TildeCode{8}
104 \def\EncodedTilde{\EscapeChar\TildeCode}
105 \def\Tilde{\noexpand\SaveTilde}
106
107 % |" IS AN ENCODED \"

```

```

108 \def\UmlautCode{"}
109 \def\EncodedUmlaut{\EscapeChar\UmlautCode}
110 \def\Umlaut{\noexpand\SaveUmlaut}
111
112 % | ' IS AN ENCODED \
113 \def\GraveCode{' }
114 \def\EncodedGrave{\EscapeChar\GraveCode}
115 \def\Grave{\noexpand\SaveGrave}
116
117 % | ' IS AN ENCODED \
118 \def\AcuteCode{' }
119 \def\EncodedAcute{\EscapeChar\AcuteCode}
120 \def\Acute{\noexpand\SaveAcute}
121
122 % |= IS AN ENCODED \=
123 \def\MacronCode{=}
124 \def\EncodedMacron{\EscapeChar\MacronCode}
125 \def\Macron{\noexpand\SaveMacron}
126
127 % |. IS AN ENCODED \.
128 \def\OverdotCode{.}
129 \def\EncodedOverdot{\EscapeChar\OverdotCode}
130 \def\Overdot{\noexpand\SaveOverdot}
131
132 % |u IS AN ENCODED \u
133 \def\BreveCode{u}
134 \def\EncodedBreve{\EscapeChar\BreveCode}
135 \def\Breve{\noexpand\SaveBreve}
136
137 % |v IS AN ENCODED \v
138 \def\CaronCode{v}
139 \def\EncodedCaron{\EscapeChar\CaronCode}
140 \def\Caron{\noexpand\SaveCaron}
141
142 % |H IS AN ENCODED \H
143 \def\DoubleAcuteCode{H}
144 \def\EncodedDoubleAcute{\EscapeChar\DoubleAcuteCode}
145 \def\DoubleAcute{\noexpand\SaveDoubleAcute}
146
147 % |c IS AN ENCODED \c
148 \def\CedillaCode{c}
149 \def\EncodedCedilla{\EscapeChar\CedillaCode}
150 \def\Cedilla{\noexpand\SaveCedilla}
151
152 % |d IS AN ENCODED \d
153 \def\UnderdotCode{d}
154 \def\EncodedUnderdot{\EscapeChar\UnderdotCode}
155 \def\Underdot{.}% CANNOT \edef \d
156
157 % |t IS AN ENCODED \t

```

```

158 \def\ArchJoinCode{t}
159 \def\EncodedArchJoin{\EscapeChar\ArchJoinCode}
160 \def\ArchJoin{.}% CANNOT \edef \t
161
162 % |b IS AN ENCODED \b
163 \def\LineUnderCode{b}
164 \def\EncodedLineUnder{\EscapeChar\LineUnderCode}
165 \def\LineUnder{.}% CANNOT \edef \b
166
167 % |C IS AN ENCODED \copyright
168 \def\CopyrightCode{C}
169 \def\EncodedCopyright{\EscapeChar\CopyrightCode}
170 \def\Copyright{.}% CANNOT \edef \copyright
171
172 % |p IS AN ENCODED \pounds
173 \def\PoundsCode{p}
174 \def\EncodedPounds{\EscapeChar\PoundsCode}
175 \def\Pounds{\SavePounds}
176
177 % |[ IS AN ENCODED {
178 \def\LBCode{[}
179 \def\EncodedLB{\EscapeChar\LBCode}
180 \def\UnencodedLB{.}
181 \def\LB{\EncodedLB}
182
183 % |] IS AN ENCODED }
184 \def\RBCode{]}
185 \def\EncodedRB{\EscapeChar\RBCode}
186 \def\UnencodedRB{.}
187 \def\RB{\EncodedRB}
188
189 % |z IS AN ENCODED \dag
190 \def\DaggerCode{z}
191 \def\EncodedDagger{\EscapeChar\DaggerCode}
192 \def\Dagger{.}% CANNOT \edef \dag
193
194 % |Z IS AN ENCODED \ddag
195 \def\DoubleDaggerCode{Z}
196 \def\EncodedDoubleDagger{\EscapeChar\DoubleDaggerCode}
197 \def\DoubleDagger{.}% CANNOT \edef \ddag
198
199 % |S IS AN ENCODED \S
200 \def\SectionSymbolCode{S}
201 \def\EncodedSectionSymbol{\EscapeChar\SectionSymbolCode}
202 \def\SectionSymbol{\SaveSectionSymbol}
203
204 % |P IS AN ENCODED \P
205 \def\PilcrowCode{P}
206 \def\EncodedPilcrow{\EscapeChar\PilcrowCode}
207 \def\Pilcrow{.}% CANNOT \edef \P

```

```

208
209 % |E IS AN ENCODED \AE
210 \def\AEscCode{E}
211 \def\EncodedAEsc{\EscapeChar\AEscCode}
212 \def\AEsc{\SaveAEsc}
213
214 % |e IS AN ENCODED \ae
215 \def\aesCode{e}
216 \def\Encodedaes{\EscapeChar\aesCode}
217 \def\aes{\Saveaes}
218
219 % |O IS AN ENCODED \OE
220 \def\OEthelCode{O}
221 \def\EncodedOEthel{\EscapeChar\OEthelCode}
222 \def\OEthel{\SaveOEthel}
223
224 % |o IS AN ENCODED \oe
225 \def\oethelCode{o}
226 \def\Encodedoethel{\EscapeChar\oethelCode}
227 \def\oethel{\Saveoethel}
228
229 % |A IS AN ENCODED \AA
230 \def\AngstromCode{A}
231 \def\EncodedAngstrom{\EscapeChar\AngstromCode}
232 \def\Angstrom{\SaveAngstrom}
233
234 % |a IS AN ENCODED \aa
235 \def\angstromCode{a}
236 \def\Encodedangstrom{\EscapeChar\angstromCode}
237 \def\angstrom{\Saveangstrom}
238
239 % |Q IS AN ENCODED \O
240 \def\SlashedOCode{Q}
241 \def\EncodedSlashedO{\EscapeChar\SlashedOCode}
242 \def\SlashedO{\SaveSlashedO}
243
244 % |q IS AN ENCODED \o
245 \def\SlashedoCode{q}
246 \def\EncodedSlashedo{\EscapeChar\SlashedoCode}
247 \def\Slashedo{\SaveSlashedo}
248
249 % |L IS AN ENCODED \L
250 \def\BarredLCode{L}
251 \def\EncodedBarredL{\EscapeChar\BarredLCode}
252 \def\BarredL{\SaveBarredL}
253
254 % |l IS AN ENCODED \l
255 \def\BarredlCode{l}
256 \def\EncodedBarredl{\EscapeChar\BarredlCode}
257 \def\Barredl{\SaveBarredl}

```

```

258
259 % |s IS AN ENCODED \ss
260 \def\EszettCode{s}
261 \def\EncodedEszett{\EscapeChar\EszettCode}
262 \def\Eszett{\SaveEszett}
263
264 \newcounter{@letterindex}
265 \newcounter{@@letterindex}
266 \newcounter{@@@letterindex}
267 \newcounter{@wordindex}
268 \newcounter{@iargc}
269 \newcounter{@gobblesize}
270 \newcounter{@maxrotation}
271 \newcounter{@stringsize}
272 \newcounter{@@stringsize}
273 \newcounter{@revisedstringsize}
274 \newcounter{@gobbleindex}
275 \newcounter{@charsfound}
276 \newcounter{@alph}
277 \newcounter{@alphaindex}
278 \newcounter{@capstrigger}
279 \newcounter{@fromindex}
280 \newcounter{@toindex}
281 \newcounter{@previousindex}
282 \newcounter{@flag}
283 \newcounter{@matchloc}
284 \newcounter{@matchend}
285 \newcounter{@matchsize}
286 \newcounter{@matchmax}
287 \newcounter{@skipped}

288 %%%% 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.

```

289 \newcommand\defaultTreatments{%
290   \def\EncodingTreatment{v}% <--Set=v to decode special chars (vs. q,e)
291   \def\AlphaCapsTreatment{1}% <--Set=1 to retain uppercase (vs. 0,2)
292   \def\AlphaTreatment{1}% <--Set=1 to retain lowercase (vs. 0,2)
293   \def\PunctuationTreatment{1}% <--Set=1 to retain punctuation (vs. 0)
294   \def\NumeralTreatment{1}% <--Set=1 to retain numerals (vs. 0)
295   \def\SymbolTreatment{1}% <--Set=1 to retain special chars (vs. 0)
296   \def\BlankTreatment{1}% <--Set=1 to retain blanks (vs. 0)
297   \def\CapitalizeString{0}% <--Set=0 for no special action (vs. 1,2)
298   \def\SeekBlankSpace{0}% <--Set=0 for no special action (vs. 1,2)
299 }
300 \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.

```

301 % QUICK WAY TO SET UP TREATMENTS BY WHICH \@rotate HANDLES VARIOUS
302 % CHARACTERS
303 \newcommand\Treatments[6]{%
304   \def\AlphaCapsTreatment{#1}% <--Set=0 to remove uppercase
305   %                               =1 to retain uppercase
306   %                               =2 to change UC to lc
307   \def\AlphaTreatment{#2}% <--Set=0 to remove lowercase
308   %                               =1 to retain lowercase
309   %                               =2 to change lc to UC
310   \def\PunctuationTreatment{#3}%<--Set=0 to remove punctuation
311   %                               =1 to retain punctuation
312   \def\NumeralTreatment{#4}% <--Set=0 to remove numerals
313   %                               =1 to retain numerals
314   \def\SymbolTreatment{#5}% <--Set=0 to remove special chars
315   %                               =1 to retain special chars
316   \def\BlankTreatment{#6}% <--Set=0 to remove blanks
317   %                               =1 to retain blanks
318 }

```

`\+` 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.

```

319 % REENCODE MULTIBYTE SYMBOLS USING THE stringstrings ENCODING METHOD
320 \newcommand\+{%
321   \def\${\EncodedDollar}%
322   \def~{\EncodedBlankSpace}%
323   \def~{\EncodedCircumflex}%
324   \def~{\EncodedTilde}%
325   \def"\{\EncodedUmlaut}%
326   \def'\{\EncodedGrave}%
327   \def'\{\EncodedAcute}%
328   \def=\{\EncodedMacron}%
329   \def.\{\EncodedOverdot}%
330   \def\u{\EncodedBreve}%
331   \def\v{\EncodedCaron}%
332   \def\H{\EncodedDoubleAcute}%
333   \def\c{\EncodedCedilla}%
334   \def\d{\EncodedUnderdot}%
335   \def\t{\EncodedArchJoin}%
336   \def\b{\EncodedLineUnder}%
337   \def\copyright{\EncodedCopyright}%
338   \def\pounds{\EncodedPounds}%
339   \def\{\{\EncodedLeftBrace}%
340   \def\}\{\EncodedRightBrace}%

```



```

341 \def\_{\EncodedUnderscore}%
342 \def\dag{\EncodedDagger}%
343 \def\ddag{\EncodedDoubleDagger}%
344 \def\S{\EncodedSectionSymbol}%
345 \def\P{\EncodedPilcrow}%
346 \def\AE{\EncodedAEsc}%
347 \def\ae{\Encodedaesc}%
348 \def\OE{\EncodedOEthel}%
349 \def\oe{\Encodedoethel}%
350 \def\AA{\EncodedAngstrom}%
351 \def\aa{\Encodedangstrom}%
352 \def\O{\EncodedSlashedO}%
353 \def\o{\EncodedSlashedo}%
354 \def\L{\EncodedBarredL}%
355 \def\l{\EncodedBarredl}%
356 \def\ss{\EncodedEszett}%
357 }

```

\? The command \? reverts the character encodings back to the standard L^AT_EX definitions. The command effectively undoes a previously enacted \+.

```

358 % WHEN TASK IS DONE, REVERT ENCODING TO STANDARD ENCODING METHOD
359 \newcommand\?{%
360 \let\$SaveDollar%
361 \let~SaveHardspace%
362 \let^SaveCircumflex%
363 \let~SaveTilde%
364 \let"\SaveUmlaut%
365 \let'\SaveGrave%
366 \let'\SaveAcute%
367 \let=\SaveMacron%
368 \let.\SaveOverdot%
369 \let\uSaveBreve%
370 \let\vSaveCaron%
371 \let\HSaveDoubleAcute%
372 \let\cSaveCedilla%
373 \let\dSaveUnderdot%
374 \let\tSaveArchJoin%
375 \let\bSaveLineUnder%
376 \let\copyrightSaveCopyright%
377 \let\poundsSavePounds%
378 \let\{\SaveLeftBrace%
379 \let\}\SaveRightBrace%
380 \let\_SaveUnderscore%
381 \let\dagSaveDagger%
382 \let\ddagSaveDoubleDagger%
383 \let\SSaveSectionSymbol%
384 \let\PSavePilcrow%
385 \let\AESaveAEsc%
386 \let\aeSaveaesc%

```

```

387 \let\OE\SaveOEthel%
388 \let\oe\Saveoethel%
389 \let\AA\SaveAngstrom%
390 \let\aa\Saveangstrom%
391 \let\O\SaveSlashedO%
392 \let\o\SaveSlashedo%
393 \let\L\SaveBarredL%
394 \let\l\SaveBarredl%
395 \let\ss\SaveEszett%
396 }

```

```

397 %%%% 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.

```

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

```

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

```

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

```

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

```

400 \@decodepointer{#3}%
401 \setcounter{@fromindex}{\@fromtoindex}%
402 \@decodepointer{#4}%
403 \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.

```

404 \setcounter{@gobblesize}{\value{@stringsize}}%
405 \ifthenelse{\value{@toindex} > \value{@stringsize}}%

```

```

406     {\setcounter{@maxrotation}{\value{@stringsize}}}%
407     {\setcounter{@maxrotation}{\value{@toindex}}}%
408 \addtocounter{@gobblesize}{-\value{@maxrotation}}%
409 \addtocounter{@gobblesize}{\value{@fromindex}}%
410 \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.

```

411 \setcounter{@letterindex}{0}%
412 \edef\rotatingword{#2}%
413 \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.

```

414 \if 0\CapitalizeString%
415 % DO NOT SET CAPITALIZE TRIGGER FOR FIRST CHARACTER
416 \setcounter{@capstrigger}{0}%
417 \else
418 % SAVE CERTAIN TREATMENTS FOR LATER RESTORATION
419 \let\SaveAlphaTreatment\AlphaTreatment%
420 \let\SaveAlphaCapsTreatment\AlphaCapsTreatment%
421 % SET CAPITALIZE TRIGGER FOR FIRST CHARACTER
422 \setcounter{@capstrigger}{1}%
423 \@forcecapson%
424 \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.*).

```

425 \@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.

```

426 \whiledo{\value{@letterindex} < \value{@maxrotation}}{%
427     \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.

```

428 % IF NEXT CHARACTER BLANK WHILE \CapitalizeString=2,
429 % SET OR KEEP ALIVE TRIGGER.

```

```

430 \if 2\CapitalizeString%
431 \isnextbyte[q]{\EncodedBlankSpace}{\rotatingword}%
432 \if F\theresult\isnextbyte[q]{\BlankSpace}{\rotatingword}\fi%
433 \if T\theresult%
434 \if 0\arabic{@capstrigger}%
435 \@forcecapson%
436 \@defineactions%
437 \fi
438 \setcounter{@capstrigger}{2}%
439 \fi
440 \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.

```

441 % CHECK IF NEXT CHARACTER IS A SYMBOL
442 \isnextbyte[q]{\EscapeChar}{\rotatingword}%
443 \ifthenelse{\value{@letterindex} < \value{@fromindex}}{%
444   {%
445 % THIS CHARACTER WILL EVENTUALLY BE GOBBLED
446 \if T\theresult%
447 % ROTATE THE ESCAPE CHARACTER, WHICH WILL LEAVE THE SYMBOL ENCODED
448 % FOR PROPER GOBBLING (ESCAPE CHARACTER DOESN'T COUNT AS A LETTER)
449 \edef\rotatingword{\@rotate{\rotatingword}}%
450 \addtocounter{@letterindex}{-1}%
451 \else
452 % NORMAL CHARACTER OR SYMBOL CODE... ROTATE IT
453 \edef\rotatingword{\@rotate{\rotatingword}}%
454 \fi
455 }%
456 {%
457 % THIS CHARACTER WILL EVENTUALLY MAKE IT INTO SUBSTRING
458 \if T\theresult%
459 % ROTATE THE SYMBOL USING DEFINED TREATMENT RULES
460 \edef\rotatingword{\ESCrotate{\expandafter\@gobble\rotatingword}}%
461 \else
462 % NORMAL CHARACTER... ROTATE IT
463 \edef\rotatingword{\@rotate{\rotatingword}}%
464 \fi
465 }%

```

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.

```

466 %   DECREMENT CAPITALIZATION TRIGGER TOWARDS 0, EVERY TIME THROUGH LOOP
467   \if 0\arabic{@capstrigger}%
468   \else
469     \addtocounter{@capstrigger}{-1}%
470     \if 0\arabic{@capstrigger}\@relaxcapson\fi
471   \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.

```

472 %   IF SOUGHT SPACE IS FOUND, END ROTATION OF STRING
473   \if 0\SeekBlankSpace\else
474     \isnextbyte[q]{\EncodedBlankSpace}{\rotatingword}%
475     \if F\theresult\isnextbyte[q]{\BlankSpace}{\rotatingword}\fi%
476     \if T\theresult%
477       \if 1\SeekBlankSpace%
478         STOP ROTATION, KEEP REMAINDER OF STRING
479         \setcounter{@maxrotation}{\value{@letterindex}}%
480       \else
481         STOP ROTATION, THROW AWAY REMAINDER OF STRING
482         \addtocounter{@gobblesize}{\value{@maxrotation}}%
483         \setcounter{@maxrotation}{\value{@letterindex}}%
484         \addtocounter{@gobblesize}{-\value{@maxrotation}}%
485       \fi
486     \fi
487   \fi
488 }%

```

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.

```

489 %   GOBBLE AWAY THAT PART OF STRING THAT ISN'T PART OF REQUESTED SUBSTRING
490   \@gobblearg{\rotatingword}{\arabic{@gobblesize}}%
491   \edef\thestring{\gobbledword}%
492   \if v#1\thestring\fi%
493 \?}

```

Many of the following commands are self-explanatory. 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

```
494 % Convert Lower to Uppercase; retain all symbols, numerals,
495 % punctuation, and blanks.
496 \newcommand\caseupper[2][v]{%
497   \Treatments{1}{2}{1}{1}{1}{1}%
498   \substring[#1]{#2}{1}{\@MAXSTRINGSIZE}%
499   \defaultTreatments%
500 }
```

\caselower

```
501 % Convert Upper to Lowercase; retain all symbols, numerals,
502 % punctuation, and blanks.
503 \newcommand\caselower[2][v]{%
504   \Treatments{2}{1}{1}{1}{1}{1}%
505   \substring[#1]{#2}{1}{\@MAXSTRINGSIZE}%
506   \defaultTreatments%
507 }
```

\solelyuppercase

```
508 % Convert Lower to Uppercase; discard symbols, numerals, and
509 % punctuation, but keep blanks.
510 \newcommand\solelyuppercase[2][v]{%
511   \Treatments{1}{2}{0}{0}{0}{1}%
512   \substring[#1]{#2}{1}{\@MAXSTRINGSIZE}%
513   \defaultTreatments%
514 }
```

\solelylowercase

```
515 % Convert Upper to Lowercase; discard symbols, numerals, and
516 % punctuation, but keep blanks.
517 \newcommand\solelylowercase[2][v]{%
518   \Treatments{2}{1}{0}{0}{0}{1}%
519   \substring[#1]{#2}{1}{\@MAXSTRINGSIZE}%
520   \defaultTreatments%
521 }
```

\change case

```
522 % Convert Lower to Uppercase & Upper to Lower; retain all symbols, numerals,
523 % punctuation, and blanks.
524 \newcommand\change case[2][v]{%
525   \Treatments{2}{2}{1}{1}{1}{1}%
526   \substring[#1]{#2}{1}{\@MAXSTRINGSIZE}%
527   \defaultTreatments%
528 }
```

\noblanks

```
529 % Remove blanks; retain all else.
530 \newcommand\noblanks[2][v]{%
531   \Treatments{1}{1}{1}{1}{1}{0}%
532   \substring[#1]{#2}{1}{\@MAXSTRINGSIZE}%
533   \defaultTreatments%
534 }
```

\nosymbolsnumerals

```
535 % Retain case; discard symbols & numerals; retain
536 % punctuation & blanks.
537 \newcommand\nosymbolsnumerals[2][v]{%
538   \Treatments{1}{1}{1}{0}{0}{1}%
539   \substring[#1]{#2}{1}{\@MAXSTRINGSIZE}%
540   \defaultTreatments%
541 }
```

\alphabetic

```
542 % Retain case; discard symbols, numerals &
543 % punctuation; retain blanks.
544 \newcommand\alphabetic[2][v]{%
545   \Treatments{1}{1}{0}{0}{0}{1}%
546   \substring[#1]{#2}{1}{\@MAXSTRINGSIZE}%
547   \defaultTreatments%
548 }
```

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

```
549 % Capitalize first character of string,
550 \newcommand\capitalize[2][v]{%
551   \defaultTreatments%
552   \def\CapitalizeString{1}%
553   \substring[#1]{#2}{1}{\@MAXSTRINGSIZE}%
554   \def\CapitalizeString{0}%
555 }
```

\capitalizewords

```
556 % Capitalize first character of each word in string,
557 \newcommand\capitalizewords[2][v]{%
558   \defaultTreatments%
559   \def\CapitalizeString{2}%
560   \substring[#1]{#2}{1}{\@MAXSTRINGSIZE}%
561   \def\CapitalizeString{0}%
562 }
```

`\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`.

```

563 % REVERSES SEQUENCE OF CHARACTERS IN STRING
564 \newcommand\reversestring[2][v]{%
565   \def\@reversedstring{}%
566   \+\@getstringlength{#2}{\@stringsize}\?%
567   \setcounter{@@@letterindex}{\the@@stringsize}%
568   \whiledo{\the@@@letterindex > 0}{%
569     \if e#1%
570       \substring[e]{#2}{\the@@@letterindex}{\the@@@letterindex}%
571     \else
572       \substring[q]{#2}{\the@@@letterindex}{\the@@@letterindex}%
573     \fi
574     \edef\@reversedstring{\@reversedstring\thestring}%
575     \addtocounter{@@@letterindex}{-1}%
576   }%
577   \edef\thestring{\@reversedstring}%
578   \if v#1\thestring\fi%
579 }

```

`\convertchar` Takes a string, and replaces each occurrence 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.

```

580 % TAKES A STARTING STRING #2 AND SUBSTITUTES A SPECIFIED STRING #4
581 % FOR EVERY OCCURANCE OF A PARTICULAR GIVEN CHARACTER #3. THE
582 % CHARACTER TO BE CONVERTED MAY BE EITHER A PLAIN CHARACTER OR
583 % AN ENCODABLE SYMBOL.
584 \newcommand\convertchar[4][v]{%
585   \+%
586   \edef\encodedstring{#2}%
587   \edef\encodedfromarg{#3}%
588   \edef\encodedtoarg{#4}%
589   \?%
590   \isnextbyte[q]{\EscapeChar}{\encodedfromarg}%
591   \if F\theresult%
592 %   PLAIN "FROM" ARGUMENT
593     \@convertbytetostring[#1]{\encodedstring}{#3}{\encodedtoarg}%
594   \else
595 %   ENCODABLE "FROM" ARGUMENT
596     \@convertsymboltostring[#1]{\encodedstring}%
597     {\expandafter\@gobble\encodedfromarg}{\encodedtoarg}%
598   \fi
599 }

```


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

```

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

  Rotate leading blank spaces to end of string

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

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

603   \def\SeekBlankSpace{1}%

  Leave rotated characters alone

604   \Treatments{1}{1}{1}{1}{1}{1}%

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

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

  Rotate trailing spaces.

606   \@treatleadingspaces[#1]{\thestring}{}%
607   \defaultTreatments%
608 }
```

`\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.

```

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

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

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

  The Treatments are specified to remove all characters.

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

  Trailing spaces are also deleted.

615   \@treatleadingspaces[#1]{\thestring}{x}%
616   \defaultTreatments%
617 }
```

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

```

618 % GETS NEXT WORD FROM STRING #2.
619 % NOTE: ROTATES BACK TO BEGINNING, AFTER STRING OTHERWISE EXHAUSTED
```

```

620 \newcommand\getnextword[2][v]{%
621   \getaword[#1]{#2}{1}%
622 }

```

\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.

```

623 % GETS WORD #3 FROM STRING #2.
624 % NOTE: ROTATES BACK TO BEGINNING, AFTER STRING OTHERWISE EXHAUSTED
625 \newcommand\getaword[3][v]{%
626   \setcounter{@wordindex}{1}%
627   \+ \edef\thestring{#2}\?%
628   \@treatleadingspaces[e]{\thestring}{}%
629   \whiledo{\value{@wordindex} < #3}{%
630     \rotateword[e]{\thestring}%
631     \addtocounter{@wordindex}{1}%
632   }%
633   \@getnextword[#1]{\thestring}%
634 }

```

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

```

635 \newcommand\rotateleadingspaces[2][v]{%
636   \@treatleadingspaces[#1]{#2}{}%
637 }

```

\removeleadingspaces Remove leading spaces from string #2.

```

638 \newcommand\removeleadingspaces[2][v]{%
639   \@treatleadingspaces[#1]{#2}{x}%
640 }

```

\stringencode

```

641 % ENCODE STRING; UNLIKE OTHER COMMANDS, DEFAULT IS NO PRINT
642 \newcommand\stringencode[2][e]{%
643   \defaultTreatments%
644   \substring[#1]{#2}{1}{\@MAXSTRINGSIZE}%
645 }

```

\stringdecode

```

646 % DECODE STRING
647 \newcommand\stringdecode[2][v]{%
648   \defaultTreatments%
649   \substring[#1]{#2}{1}{\@MAXSTRINGSIZE}%
650 }

```

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

```
651 % SINGLE-CHARACTER VERSION OF \gobblechars. IN THIS CASE, TWO-BYTE
652 % ESCAPE SEQUENCES, WHEN ENCOUNTERED, COUNT AS A SINGLE GOBBLE.
653 \newcommand\gobblechar[2][q]{\+%
654   \@gobblearg{#2}{1}%
655   \?\retokenize[#1]{\gobbledword}%
656 }
```

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

```
657 % USER CALLABLE VERSION OF \@gobblearg. TURNS ON REENCODING.
658 % GOBBLE FIRST #3 CHARACTERS FROM STRING #2. TWO-BYTE
659 % ESCAPE SEQUENCES, WHEN ENCOUNTERED, COUNT AS A SINGLE GOBBLE.
660 \newcommand\gobblechars[3][q]{\+%
661   \@gobblearg{#2}{#3}%
662   \?\retokenize[#1]{\gobbledword}%
663 }
```

`\retokenize` One of the key `stringstrings` routines that provides several indispensable functions. Its function is to take an encoded string #2 that has been given, and repopulate the string with its L^AT_EX 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 2 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{\buildtoken\inexttoken}
\def\buildtokenii{\buildtokeni\iinexttoken}
\def\buildtokeniii{\buildtokenii\iiinexttoken}
\def\buildtokeniv{\buildtokeniii\ivnexttoken}
```

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

```
664 % CONVERTS ENCODED STRING BACK INTO TOKENIZED FORM (i.e., def'ED).
665 \newcommand\retokenize[2][q]{\+%
666   \edef\@svstring{#2}%
667   \edef\buildtoken{}%
668   \@getstringlength{#2}{\@stringsize}\?%
669   \setcounter{\@letterindex}{0}%
```

```

670 \whiledo{\the@@letterindex < \the@@stringsize}{%
671   \setcounter{@previousindex}{\the@@letterindex}%
672   \addtocounter{@@letterindex}{1}%
673   \substring[e]{\@svstring}{\the@@letterindex}{\the@@letterindex}%
674   \@retokenizechar{\thestring}{\roman{@@letterindex}nexttoken}%
675   \expandafter\def\csname buildtoken\roman{@@letterindex}%
676     \expandafter\endcsname\expandafter%
677     {\csname buildtoken\roman{@previousindex}\expandafter\endcsname%
678       \csname\roman{@@letterindex}nexttoken\endcsname}%
679 }%
680 \expandafter\def\expandafter\buildtokenq\expandafter{%
681   \csname buildtoken\roman{@@letterindex}\endcsname}%
682 \def\thestring{\buildtokenq}%
683 \if v#1\thestring\fi
684 }

685 % %%%% 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*.

```

686 % USER CALLABLE VERSION of \@getstringlength:
687 % GET'S STRING LENGTH OF [#2], PUTS RESULT IN \theresult. PRINTS RESULT
688 % UNLESS IN QUIET [q] MODE.
689 \newcommand\stringlength[2][v]{\+%
690   \@getstringlength{#2}{@@stringsize}%
691   \edef\theresult{\arabic{@@stringsize}}%
692   \if v#1\theresult\fi%
693 \?}

```

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

```

694 % CHECKS TO SEE IF THE CHARACTER [#3] APPEARS ANYWHERE IN STRING [#2].
695 % THE NUMBER OF OCCURANCES IS PRINTED OUT, EXCEPT WHEN [#1]=q, QUIET
696 % MODE. RESULT IS ALSO STORED IN \theresult . TO FIND SPACES, ARG3
697 % SHOULD BE SET TO {~}, NOT { }.
698 \newcommand\findchars[3][v]{\+%
699   \@getstringlength{#2}{@@stringsize}%
700   \setcounter{@charsfound}{0}%
701   \setcounter{@@letterindex}{0}%

```

Loop through each character of #2.

```

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

```

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

```

704     \testmatchingchar{#2}{\arabic{@@letterindex}}{#3}%
705     \ifmatchingchar\addtocounter{@charsfound}{1}\fi
706 }%
707 \edef\theresult{\arabic{@charsfound}}%
708 \if q#1\else\theresult\fi%
709 \?}

```

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

```

710 % CHECKS TO FIND LOCATION OF FIRST OCCURANCE OF [#3] IN STRING [#2].
711 % THE LOCATION IS PRINTED OUT, EXCEPT WHEN [#1]=q, QUIET
712 % MODE. RESULT IS ALSO STORED IN \theresult . TO FIND SPACES, ARG3
713 % SHOULD BE SET TO {~}, NOT { }.
714 \newcommand\whereischar[3][v]{\+%
715   \@getstringlength{#2}{@stringsize}%
716   \edef\@theresult{0}%
717   \setcounter{@@letterindex}{0}%

```

Loop through characters of #2 sequentially.

```

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

```

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

```

720     \testmatchingchar{#2}{\arabic{@@letterindex}}{#3}%
721     \ifmatchingchar%
722       \edef\@theresult{\the@@letterindex}%
723       \setcounter{@@letterindex}{\the@stringsize}%
724     \fi
725   }%
726   \edef\theresult{\@theresult}%
727   \if q#1\else\theresult\fi%
728 \?}

```

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

```

729 % LIKE \whereischar, EXCEPT FOR WORDS
730 \newcommand\whereisword[3][v]{\+%
731   \setcounter{@skipped}{0}%

```

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

```

732   \edef\@@@teststring{#2}%
733   \edef\@matchstring{#3}%
734   \@getstringlength{#2}{@stringsize}%
735   \@getstringlength{#3}{@matchsize}%
736   \setcounter{@matchmax}{\the@stringsize}%
737   \addtocounter{@matchmax}{-\the@matchsize}%

```

```

738 \addtocounter{@matchmax}{1}%
739 \setcounter{@flag}{0}%

```

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

```

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

```

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

```

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

```

If none found, we are done.

```

746 {%
747 \setcounter{@flag}{1}%
748 }%

```

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.

```

749 {%
750 \setcounter{@matchend}{\theresult}%
751 \addtocounter{@matchend}{\value{@matchsize}}%
752 \addtocounter{@matchend}{-1}%
753 \substring[e]{\@@@teststring}{\the@matchloc}{\the@matchend}%
754 \ifthenelse{\equal{\thestring}{\@matchstring}}{%

```

Found a match! Save the match location

```

755 {%
756 \setcounter{@flag}{1}%
757 \addtocounter{@matchloc}{\the@skipped}%
758 \edef\theresult{\the@matchloc}%
759 }%

```

False alarm. Determine if lopping off the leading characters of `\@@@teststring` (to discard the false-alarm occurrence) 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 occurrences 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 occurrence 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}%
792         \addtocounter{@matchloc}{\the@matchsize}%
793         \ifthenelse{\the@matchloc > \the@@stringsize}%
794         {%

```

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

```

795         \edef\theresult{\the@charsfound}%
796         \setcounter{@charsfound}{-1}%
797     }%
798     {%

```

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

```

799         \substring[e]{\@@@teststring}{\the@matchloc}{\@MAXSTRINGSIZE}%

```

```

800         \edef\@@teststring{\thestring}%
801         \edef\@@@teststring{\@@teststring}%
802     }%
803 }%
804 }%
805 \if q#1\else\theresult\fi%
806 }

```

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

```

807 \newcounter{@matchloc}
808 % LIKE \convertchar, EXCEPT FOR WORDS
809 \newcommand\convertword[4][v]{%
810     \+\edef\@@teststring{#2}%
811     \edef\@fromstring{#3}%
812     \edef\@tostring{#4}\?%
813     \edef\@@@teststring{\@@teststring}%
814     \def\@buildfront{%
815         \edef\@buildstring{\@@teststring}%
816         \setcounter{@charsfound}{0}%
817         \whiledo{\the@charsfound > -1}{%

```

Seek occurrence of \@fromstring in larger \@@teststring

```

818 \whereisword[q]{\@@teststring}{\@fromstring}%
819     \setcounter{@matchloc}{\theresult}%
820     \ifthenelse{\the@matchloc = 0}%
821     {%

```

Not found. Done.

```

822         \setcounter{@charsfound}{-1}%
823     }%
824     {%

```

Potential matchstring.

```

825         \addtocounter{@charsfound}{1}%

```

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

```

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

```

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

```

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

```

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

```

829         \addtocounter{@matchloc}{1}%

```



```

830      \addtocounter{@matchloc}{\the@matchsize}%
831      \ifthenelse{\the@matchloc > \the@stringsize}%
832      {%

```

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

```

833      \setcounter{@charsfound}{-1}%
834      \edef\@buildstring{\@buildfront\@tostring}%
835      }%
836      {%

```

...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.

```

837      \substring[e]{\@@teststring}{\the@matchloc}{\@MAXSTRINGSIZE}%
838      \edef\@@teststring{\thestring}%
839      \edef\@@@teststring{\@@teststring}%
840      \edef\@buildstring{\@buildfront\@tostring\@@@teststring}%
841      \edef\@buildfront{\@buildfront\@tostring}%
842      }%
843      }%
844      }%
845      \substring[#1]{\@buildstring}{1}{\@MAXSTRINGSIZE}%
846      }

```

\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.

```

847 % WORD COUNT
848 \newcommand\wordcount[2][v]{\+%
849   \edef\@argv{\#2}
850   \@getstringlength{\@argv}{@stringsize}
851   \setcounter{@iargc}{0}
852   \whiledo{\value{@stringsize} > 0}{%
853     \addtocounter{@iargc}{1}%
854     \removeword[e]{\@argv}%
855     \edef\@argv{\thestring}%
856     \@getstringlength{\@argv}{@stringsize}%
857   }
858   \edef\theresult{\arabic{@iargc}}%
859   \if v#1\theresult\fi%
860 \?}

```

\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.

```

861 % OBTAINS ARGUMENTS (WORDS) IN #1 ALA UNIX getarg COMMAND
862 % narg CONTAINS NUMBER OF ARGUMENTS.  ARGUMENTS CONTAINED IN
863 % argi, argii, argiii, argiv, ETC.
864 % v mode disabled
865 \newcommand\getargs[2][q]{\+%
866   \if v#1\def\@mode{q}\else\def\@mode{#1}\fi
867   \edef\@argv{#2}%
868   \@getstringlength{\@argv}{@stringsize}%
869   \setcounter{iargc}{0}%
870   \whiledo{\value{@stringsize} > 0}{%
871     \addtocounter{iargc}{1}%
872     \getaword[\@mode]{\@argv}{1}%
873     \expandafter\edef\csname arg\roman{iargc}\endcsname{\thestring}%
874     \removeword[e]{\@argv}%
875     \edef\@argv{\thestring}%
876     \@getstringlength{\@argv}{@stringsize}%
877   }
878   \edef\narg{\arabic{iargc}}%
879 \?}

880 %%%% 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**.

```

881 \def\PrimarySignalChar{@}
882 \def\SecondarySignalChar{'}
883
884 % \isnextbyte NEEDS TO OPERATE IN RAW (SINGLE BYTE) MODE SO AS TO
885 % 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.

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

Here's the first test...

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

...and the second

```
890 \let\SignalChar\SecondarySignalChar%
891 \edef\@y{\if #2#3\else\SignalChar\fi}%
892 \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.

```
893 % BECAUSE THE METHOD ONLY PRODUCES FALSE NEGATIVES, IF RESULTS DON'T
894 % AGREE FROM USING TWO DIFFERENT SIGNAL CHARACTERS, RESULT MUST BE TRUE.
895 \ifthenelse{\equal{\@x\@y}{\@y\@x}}{
896   {\edef\theresult{\@x}}%
897 % CORRECT THE FALSE NEGATIVE
898   {\edef\theresult{T}}%
899 \if q#1\else\theresult\fi
900 }
```

`\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.

```
901 % CHECKS TO SEE IF [#2]'th CHARACTER IN STRING [#1] EQUALS [#3]
902 % RESULT STORED IN BOOLEAN \ifmatchingchar
903 \newif\ifmatchingchar
904 \newcommand\testmatchingchar[3]{%
```

Extract desired character from test string

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

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

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

Is the tested character also a multi-byte symbol?

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

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

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

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

```
914 \matchingcharfalse%
915 \fi
916 \else
```

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

```
917 \if \thestring#3\matchingchartrue\else\matchingcharfalse\fi
918 \fi
919 \?}
```

\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.

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

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

```
926 \def\AlphaCapsTreatment{2}%
927 \@defineactions%
928 \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.

```
929 \def\AlphaCapsTreatment{1}%
930 \@defineactions%
931 \edef\@stringB{\ESCrotate{\expandafter\@gobble\rotatingword}}%
932 \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.

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

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

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

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

```
938 \ifthenelse{\equal{\@stringA}{\@stringB}}%
939 {\capitalizedfalse}{\capitalizedtrue}\?%
940 \defaultTreatments%
941 }
```

\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.

```
942 \newif\ifuncapitalized
943 \newcommand\testuncapitalized[1]{\+%
944 \isnextbyte[q]{\EscapeChar}{#1}%
945 \if T\theresult%
946 \def\EncodingTreatment{e}%
947 \def\rotatingword{#1}%
948 \def\AlphaTreatment{2}%
949 \@defineactions%
950 \edef\@stringA{\ESCrotate{\expandafter\@gobble\rotatingword}}%
951 \def\AlphaTreatment{1}%
952 \@defineactions%
953 \edef\@stringB{\ESCrotate{\expandafter\@gobble\rotatingword}}%
954 \else
955 \def\AlphaTreatment{2}%
956 \edef\@stringA{\@rotate{#1}}%
957 \def\AlphaTreatment{1}%
958 \edef\@stringB{\@rotate{#1}}%
959 \fi
960 \ifthenelse{\equal{\@stringA}{\@stringB}}%
961 {\uncapitalizedfalse}{\uncapitalizedtrue}\?%
962 \defaultTreatments%
963 }
```

\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.

```
964 \newif\ifleadingalpha
```

```

965 \newcommand\testleadingalpha[1]{%
966   \testcapitalized{#1}%
967   \ifcapitalized
968     \leadingalphatrue%
969   \else
970     \testuncapitalized{#1}%
971     \ifuncapitalized
972       \leadingalphatrue%
973     \else
974       \leadingalphafalse%
975     \fi
976   \fi
977 }

```

\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 non-alphabetic characters, in which case the test results is false.

```

978 \newif\ifuppercase
979 \newcommand\testuppercase[1]{%

```

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

```

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

```

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

```

983   \def\AlphaTreatment{2}%
984   \substring[e]{#1}{1}{\@MAXSTRINGSIZE}%
985   \edef\@stringB{\thestring}%
986   \ifthenelse{\equal{\@stringA}{\@stringB}}%
987   {%

```

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.

```

988     \@getstringlength{\@stringA}{\@stringsize}%
989     \ifthenelse{\value{\@stringsize} = 0}%
990     {\uppercasefalse}{\uppercasetrue}%
991   }%

```

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

```

992   {\uppercasefalse}%
993   \defaultTreatments%
994 }

```

`\ifsolelyuppercase` Compare the original string to one made solely uppercase. If they are equal (and not composed solely of blankspaces), then the original string was solely uppercase to begin with.

```

995 \newif\ifsolelyuppercase
996 \newcommand\testsolelyuppercase[1]{%
997   \stringencode{#1}%
998   \edef\@stringA{\thestring}%
999   \solelyuppercase[e]{#1}%
1000   \edef\@stringB{\thestring}%
1001   \ifthenelse{\equal{\@stringA}{\@stringB}}%
1002   {%
1003     \noblanks[q]{\@stringA}%
1004     \@getstringlength{\thestring}{@stringsize}%
1005     \ifthenelse{\value{@stringsize} = 0}%
1006     {\solelyuppercasefalse}{\solelyuppercasetrue}%
1007   }%
1008   {\solelyuppercasefalse}%
1009   \defaultTreatments%
1010 }

```

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

```

1011 \newif\iflowercase
1012 \newcommand\testlowercase[1]{%
1013   \Treatments{1}{1}{0}{0}{0}{0}%
1014   \substring[e]{#1}{1}{\@MAXSTRINGSIZE}%
1015   \edef\@stringA{\thestring}%
1016   \def\AlphaCapsTreatment{2}%
1017   \substring[e]{#1}{1}{\@MAXSTRINGSIZE}%
1018   \edef\@stringB{\thestring}%
1019   \ifthenelse{\equal{\@stringA}{\@stringB}}%
1020   {%
1021     \@getstringlength{\@stringA}{@stringsize}%
1022     \ifthenelse{\value{@stringsize} = 0}%
1023     {\lowercasefalse}{\lowercasetrue}%
1024   }%
1025   {\lowercasefalse}%
1026   \defaultTreatments%
1027 }

```

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

```

1028 \newif\ifsolelylowercase
1029 \newcommand\testsolelylowercase[1]{%
1030   \stringencode{#1}%
1031   \edef\@stringA{\thestring}%
1032   \solelylowercase[e]{#1}%
1033   \edef\@stringB{\thestring}%

```

```

1034 \ifthenelse{\equal{\@stringA}{\@stringB}}%
1035 {%
1036   \noblanks[q]{\@stringA}%
1037   \@getstringlength{\thestring}{@stringsize}%
1038   \ifthenelse{\value{@stringsize} = 0}%
1039   {\solelylowercasefalse}{\solelylowercasetrue}%
1040 }%
1041 {\solelylowercasefalse}%
1042 \defaultTreatments%
1043 }

```

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

```

1044 \newif\ifalphabetic
1045 \newcommand\testalphabetic[1]{%
1046   \stringencode{#1}%
1047   \edef\@stringA{\thestring}%
1048   \alphabetic[e]{#1}%
1049   \edef\@stringB{\thestring}%
1050   \ifthenelse{\equal{\@stringA}{\@stringB}}%
1051   {%
1052     \noblanks[q]{\@stringA}%
1053     \@getstringlength{\thestring}{@stringsize}%
1054     \ifthenelse{\value{@stringsize} = 0}%
1055     {\alphabeticfalse}{\alphabetictrue}%
1056   }%
1057   {\alphabeticfalse}%
1058   \defaultTreatments%
1059 }

1060 %
1061 %%%% SUPPORT ROUTINES %%%%
1062 %

```

The following routines support the execution of the stringstrings package.

\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**.

```

1063 \newcommand\ESCrotate[1]{%
1064   \if\@fromcode#1\@tostring\else
1065   \if\PipeCode#1\@pipeaction\else
1066   \if\DollarCode#1\@dollaraction\else
1067   \if\CaratCode#1\@carataction\else
1068   \if\CircumflexCode#1\@circumflexaction\else
1069   \if\TildeCode#1\@tildeaction\else
1070   \if\UmlautCode#1\@umlautaction\else
1071   \if\GraveCode#1\@graveaction\else

```



```

1072         \if\AcuteCode#1\@acuteaction\else
1073         \if\MacronCode#1\@macronaction\else
1074         \if\OverdotCode#1\@overdotaction\else
1075         \if\LeftBraceCode#1\@leftbraceaction\else
1076         \if\RightBraceCode#1\@rightbraceaction\else
1077         \if\UnderscoreCode#1\@underscoreaction\else
1078         \if\DaggerCode#1\@daggeraction\else
1079         \if\DoubleDaggerCode#1\@doubledaggeraction\else
1080         \if\SectionSymbolCode#1\@sectionsymbolaction\else
1081         \if\PilcrowCode#1\@pilcrowaction\else
1082         \if\LBCode#1\@lbaction\else
1083         \if\RBCode#1\@rbaction\else
1084 \if\BreveCode#1\@breveaction\else
1085 \if\CaronCode#1\@caronaction\else
1086 \if\DoubleAcuteCode#1\@doubleacuteaction\else
1087 \if\CedillaCode#1\@cedillaaction\else
1088 \if\UnderdotCode#1\@underdotaction\else
1089 \if\ArchJoinCode#1\@archjoinaction\else
1090 \if\LineUnderCode#1\@lineunderaction\else
1091 \if\CopyrightCode#1\@copyrightaction\else
1092 \if\PoundsCode#1\@poundsaction\else
1093 \if\AEscCode#1\@AEscaction\else
1094 \if\aescCode#1\@aescaction\else
1095 \if\OEthelCode#1\@OEthelaction\else
1096 \if\oethelCode#1\@oethelaction\else
1097 \if\AngstromCode#1\@Angstromaction\else
1098 \if\angstromCode#1\@angstromaction\else
1099 \if\SlashedOCode#1\@slashedOaction\else
1100 \if\SlashedoCode#1\@slashedoaction\else
1101 \if\BarredlCode#1\@barredlaction\else
1102 \if\BarredLCode#1\@barredLaction\else
1103 \if\EszettCode#1\@eszettaction\else
1104         \expandafter\@gobble#1\undecipherable%
1105         \fi
1106         \fi
1107         \fi
1108         \fi
1109         \fi
1110         \fi
1111         \fi
1112         \fi
1113         \fi
1114         \fi
1115         \fi
1116         \fi
1117         \fi
1118         \fi
1119         \fi
1120         \fi
1121         \fi

```

```

1122     \fi
1123     \fi
1124 \fi
1125         \fi
1126         \fi
1127         \fi
1128         \fi
1129         \fi
1130         \fi
1131         \fi
1132         \fi
1133         \fi
1134         \fi
1135         \fi
1136         \fi
1137         \fi
1138         \fi
1139         \fi
1140         \fi
1141         \fi
1142         \fi
1143         \fi
1144         \fi
1145 }

```

`\@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.

```

1146 \newcommand\@getnextword[2][v]{%
1147   \defaultTreatments%
1148   \def\SeekBlankSpace{2}%
1149   \substring[#1]{#2}{1}{\@MAXSTRINGSIZE}%
1150   \def\SeekBlankSpace{0}%
1151 }

```

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

```

1152 \newcommand\@retokenizechar[2]{%
1153   \isnextbyte[q]{\EscapeChar}{#1}%
1154   \if T\theresult%
1155     \edef\@ESCcode{\expandafter\@gobble#1}%

```

```

1156 \if\PipeCode\@ESCcode%
1157 \expandafter\def\csname#2\endcsname{\Pipe}\else
1158 \if\DollarCode\@ESCcode%
1159 \expandafter\def\csname#2\endcsname{\$}\else
1160 \if\CaratCode\@ESCcode%
1161 \expandafter\def\csname#2\endcsname{\Carat}\else
1162 \if\CircumflexCode\@ESCcode%
1163 \expandafter\def\csname#2\endcsname{\^}\else
1164 \if\TildeCode\@ESCcode%
1165 \expandafter\def\csname#2\endcsname{\~}\else
1166 \if\UmlautCode\@ESCcode%
1167 \expandafter\def\csname#2\endcsname{\"}\else
1168 \if\GraveCode\@ESCcode%
1169 \expandafter\def\csname#2\endcsname{\'}\else
1170 \if\AcuteCode\@ESCcode%
1171 \expandafter\def\csname#2\endcsname{\'}\else
1172 \if\MacronCode\@ESCcode%
1173 \expandafter\def\csname#2\endcsname{\=}\else
1174 \if\OverdotCode\@ESCcode%
1175 \expandafter\def\csname#2\endcsname{\.}\else
1176 \if\LeftBraceCode\@ESCcode%
1177 \expandafter\def\csname#2\endcsname{\{}\else
1178 \if\RightBraceCode\@ESCcode%
1179 \expandafter\def\csname#2\endcsname{\}}\else
1180 \if\UnderscoreCode\@ESCcode%
1181 \expandafter\def\csname#2\endcsname{\_}\else
1182 \if\DaggerCode\@ESCcode%
1183 \expandafter\def\csname#2\endcsname{\dag}\else
1184 \if\DoubleDaggerCode\@ESCcode%
1185 \expandafter\def\csname#2\endcsname{\ddag}\else
1186 \if\SectionSymbolCode\@ESCcode%
1187 \expandafter\def\csname#2\endcsname{\S}\else
1188 \if\PilcrowCode\@ESCcode%
1189 \expandafter\def\csname#2\endcsname{\P}\else
1190 \if\LBCode\@ESCcode%
1191 \expandafter\def\csname#2\endcsname{\SaveLB}\else
1192 \if\RBCode\@ESCcode%
1193 \expandafter\def\csname#2\endcsname{\SaveRB}\else
1194 \if\BreveCode\@ESCcode\expandafter\def\csname#2\endcsname{\u}\else
1195 \if\CaronCode\@ESCcode\expandafter\def\csname#2\endcsname{\v}\else
1196 \if\DoubleAcuteCode\@ESCcode\expandafter\def\csname#2\endcsname{\H}\else
1197 \if\CedillaCode\@ESCcode\expandafter\def\csname#2\endcsname{\c}\else
1198 \if\UnderdotCode\@ESCcode\expandafter\def\csname#2\endcsname{\d}\else
1199 \if\ArchJoinCode\@ESCcode\expandafter\def\csname#2\endcsname{\t}\else
1200 \if\LineUnderCode\@ESCcode\expandafter\def\csname#2\endcsname{\b}\else
1201 \if\CopyrightCode\@ESCcode\expandafter\def\csname#2\endcsname{\copyright}\else
1202 \if\PoundsCode\@ESCcode\expandafter\def\csname#2\endcsname{\pounds}\else
1203 \if\AEscCode\@ESCcode\expandafter\def\csname#2\endcsname{\AE}\else
1204 \if\aesCode\@ESCcode\expandafter\def\csname#2\endcsname{\ae}\else
1205 \if\OEthelCode\@ESCcode\expandafter\def\csname#2\endcsname{\OE}\else

```

```

1206     \if\oethelCode\@ESCcode\expandafter\def\csname#2\endcsname{\oe}\else
1207     \if\AngstromCode\@ESCcode\expandafter\def\csname#2\endcsname{\AA}\else
1208     \if\angstromCode\@ESCcode\expandafter\def\csname#2\endcsname{\aa}\else
1209     \if\SlashedOCode\@ESCcode\expandafter\def\csname#2\endcsname{\O}\else
1210     \if\SlashedoCode\@ESCcode\expandafter\def\csname#2\endcsname{\o}\else
1211     \if\BarredlCode\@ESCcode\expandafter\def\csname#2\endcsname{\l}\else
1212     \if\BarredLCode\@ESCcode\expandafter\def\csname#2\endcsname{\L}\else
1213     \if\EszettCode\@ESCcode\expandafter\def\csname#2\endcsname{\ss}\else
1214 \expandafter\def\csname#2\endcsname{\undecipherable}%
1215 \fi
1216 \fi
1217 \fi
1218 \fi
1219 \fi
1220 \fi
1221 \fi
1222 \fi
1223 \fi
1224 \fi
1225 \fi
1226 \fi
1227 \fi
1228 \fi
1229 \fi
1230 \fi
1231 \fi
1232 \fi
1233 \fi
1234 \fi
1235 \fi
1236 \fi
1237 \fi
1238 \fi
1239 \fi
1240 \fi
1241 \fi
1242 \fi
1243 \fi
1244 \fi
1245 \fi
1246 \fi
1247 \fi
1248 \fi
1249 \fi
1250 \fi
1251 \fi
1252 \fi
1253 \fi
1254 \else
1255 \expandafter\ifx\expandafter\&#1%

```

```

1256             \expandafter\def\csname#2\endcsname{\&}\else
1257 \expandafter\ifx\expandafter\%#1%
1258             \expandafter\def\csname#2\endcsname{\%}\else
1259 \expandafter\ifx\expandafter\##1%
1260             \expandafter\def\csname#2\endcsname{\#}\else
1261 \if\EncodedBlankSpace#1\expandafter\def\csname#2\endcsname{\~}\else
1262 \expandafter\edef\csname#2\endcsname{#1}%
1263 \fi
1264 \fi
1265 \fi
1266 \fi
1267 \fi
1268 }

```

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

```

1269 % \@blankaction AND OTHER ...action'S ARE SET, DEPENDING ON VALUES OF
1270 % TREATMENT FLAGS. CHARS ARE EITHER ENCODED, DECODED, OR REMOVED.
1271 \newcommand\@defineactions{%
1272 % SET UP TREATMENT FOR SPACES, ENCODED SPACES, AND [REENCODED] SYMBOLS
1273 \if e\EncodingTreatment%
1274 % ENCODE SPACES, KEEP ENCODED SPACES ENCODED, ENCODE SYMBOLS.
1275 \edef\@blankaction{\EncodedBlankSpace}%
1276 \def\@dollaraction{\EncodedDollar}%
1277 \def\@pipeaction{\EncodedPipe}%
1278 \def\@carataction{\EncodedCarat}%
1279 \def\@circumflexaction{\EncodedCircumflex}%
1280 \def\@tildeaction{\EncodedTilde}%
1281 \def\@umlautaction{\EncodedUmlaut}%
1282 \def\@graveaction{\EncodedGrave}%
1283 \def\@acuteaction{\EncodedAcute}%
1284 \def\@macronaction{\EncodedMacron}%
1285 \def\@overdotaction{\EncodedOverdot}%
1286 \def\@breveaction{\EncodedBreve}%
1287 \def\@caronaction{\EncodedCaron}%
1288 \def\@doubleacuteaction{\EncodedDoubleAcute}%
1289 \def\@cedillaaction{\EncodedCedilla}%
1290 \def\@underdotaction{\EncodedUnderdot}%
1291 \def\@archjoinaction{\EncodedArchJoin}%
1292 \def\@lineunderaction{\EncodedLineUnder}%
1293 \def\@copyrightaction{\EncodedCopyright}%
1294 \def\@poundsaction{\EncodedPounds}%
1295 \def\@leftbraceaction{\EncodedLeftBrace}%
1296 \def\@rightbraceaction{\EncodedRightBrace}%
1297 \def\@underscoreaction{\EncodedUnderscore}%
1298 \def\@daggeraction{\EncodedDagger}%
1299 \def\@doubledaggeraction{\EncodedDoubleDagger}%
1300 \def\@sectionsymbolaction{\EncodedSectionSymbol}%

```

```

1301 \def\@pilcrowaction{\EncodedPilcrow}%
1302 \def\@eszettaction{\EncodedEszett}%
1303 \def\@lbaction{\EncodedLB}%
1304 \def\@rbaction{\EncodedRB}%
1305 \if 2\AlphaCapsTreatment%
1306 \def\@AEscaction{\Encodedaesc}%
1307 \def\@OEthelaction{\Encodedoethel}%
1308 \def\@Angstromaction{\Encodedangstrom}%
1309 \def\@slashedOaction{\EncodedSlashedO}%
1310 \def\@barredLaction{\EncodedBarredl}%
1311 \else
1312 \def\@AEscaction{\EncodedAEsc}%
1313 \def\@OEthelaction{\EncodedOEthel}%
1314 \def\@Angstromaction{\EncodedAngstrom}%
1315 \def\@slashedOaction{\EncodedSlashedO}%
1316 \def\@barredLaction{\EncodedBarredL}%
1317 \fi
1318 \if 2\AlphaTreatment%
1319 \def\@aescaction{\EncodedAEsc}%
1320 \def\@oethelaction{\EncodedOEthel}%
1321 \def\@angstromaction{\EncodedAngstrom}%
1322 \def\@slashedoaction{\EncodedSlashedO}%
1323 \def\@barredlaction{\EncodedBarredL}%
1324 \else
1325 \def\@aescaction{\Encodedaesc}%
1326 \def\@oethelaction{\Encodedoethel}%
1327 \def\@angstromaction{\Encodedangstrom}%
1328 \def\@slashedoaction{\EncodedSlashedo}%
1329 \def\@barredlaction{\EncodedBarredl}%
1330 \fi
1331 \else
1332 % EncodingTreatment=v or q:
1333 % LEAVE SPACES ALONE; RESTORE ENCODED SPACES AND SYMBOLS
1334 \def\@blankaction{\BlankSpace}%
1335 \def\@dollaraction{\Dollar}%
1336 \def\@pipeaction{\Pipe}%
1337 \def\@carataction{\Carat}%
1338 \def\@circumflexaction{\Circumflex}%
1339 \def\@tildeaction{\Tilde}%
1340 \def\@umlautaction{\Umlaut}%
1341 \def\@graveaction{\Grave}%
1342 \def\@acuteaction{\Acute}%
1343 \def\@macronaction{\Macron}%
1344 \def\@overdotaction{\Overdot}%
1345 \def\@breveaction{\Breve}%
1346 \def\@caronaction{\Caron}%
1347 \def\@doubleacuteaction{\DoubleAcute}%
1348 \def\@cedillaaction{\Cedilla}%
1349 \def\@underdotaction{\Underdot}%
1350 \def\@archjoinaction{\ArchJoin}%

```

```

1351 \def\@lineunderaction{\LineUnder}%
1352 \def\@copyrightaction{\Copyright}%
1353 \def\@poundsaction{\Pounds}%
1354 \def\@leftbraceaction{\LeftBrace}%
1355 \def\@rightbraceaction{\RightBrace}%
1356 \def\@underscoreaction{\Underscore}%
1357 \def\@daggeraction{\Dagger}%
1358 \def\@doubledaggeraction{\DoubleDagger}%
1359 \def\@sectionsymbolaction{\SectionSymbol}%
1360 \def\@pilcrowaction{\Pilcrow}%
1361 \def\@eszettaction{\Eszett}%
1362 \def\@lbaction{\UnencodedLB}%
1363 \def\@rbaction{\UnencodedRB}%
1364 \if 2\AlphaCapsTreatment%
1365 \def\@AEscaction{\aesc}%
1366 \def\@OEthelaction{\oethel}%
1367 \def\@Angstromaction{\angstrom}%
1368 \def\@slashedOaction{\Slashedo}%
1369 \def\@barredLaction{\Barredl}%
1370 \else
1371 \def\@AEscaction{\AEsc}%
1372 \def\@OEthelaction{\OEthel}%
1373 \def\@Angstromaction{\Angstrom}%
1374 \def\@slashedOaction{\SlashedO}%
1375 \def\@barredLaction{\BarredL}%
1376 \fi
1377 \if 2\AlphaTreatment%
1378 \def\@aescaction{\AEsc}%
1379 \def\@oethelaction{\OEthel}%
1380 \def\@angstromaction{\Angstrom}%
1381 \def\@slashedoaction{\SlashedO}%
1382 \def\@barredlaction{\BarredL}%
1383 \else
1384 \def\@aescaction{\aesc}%
1385 \def\@oethelaction{\oethel}%
1386 \def\@angstromaction{\angstrom}%
1387 \def\@slashedoaction{\Slashedo}%
1388 \def\@barredlaction{\Barredl}%
1389 \fi
1390 \fi
1391 % REMOVE SPACES AND ENCODED SPACES?
1392 \if 0\BlankTreatment%
1393 \edef\@blankaction{}%
1394 \fi
1395 % REMOVE ENCODED SYMBOLS?
1396 \if 0\SymbolTreatment%
1397 \def\@dollaraction{}%
1398 \def\@pipeaction{}%
1399 \def\@carataction{}%
1400 \def\@circumflexaction{}%

```

```

1401 \def\@tildeaction{}%
1402 \def\@umlautaction{}%
1403 \def\@graveaction{}%
1404 \def\@acuteaction{}%
1405 \def\@macronaction{}%
1406 \def\@overdotaction{}%
1407 \def\@breveaction{}%
1408 \def\@caronaction{}%
1409 \def\@doubleacuteaction{}%
1410 \def\@cedillaaction{}%
1411 \def\@underdotaction{}%
1412 \def\@archjoinaction{}%
1413 \def\@lineunderaction{}%
1414 \def\@copyrightaction{}%
1415 \def\@poundsaction{}%
1416 \def\@leftbraceaction{}%
1417 \def\@rightbraceaction{}%
1418 \def\@underscoreaction{}%
1419 \def\@daggeraction{}%
1420 \def\@doubledaggeraction{}%
1421 \def\@sectionsymbolaction{}%
1422 \def\@pilcrowaction{}%
1423 \def\@lbaction{}%
1424 \def\@rbaction{}%
1425 \fi
1426 % REMOVE ENCODED ALPHACAPS?
1427 \if 0\AlphaCapsTreatment%
1428 \def\@AEscaction{}%
1429 \def\@OEthelaction{}%
1430 \def\@Angstromaction{}%
1431 \def\@slashedOaction{}%
1432 \def\@barredLaction{}%
1433 \fi
1434 % REMOVE ENCODED ALPHA?
1435 \if 0\AlphaTreatment%
1436 \def\@aescaction{}%
1437 \def\@oethelaction{}%
1438 \def\@angstromaction{}%
1439 \def\@slashedoaction{}%
1440 \def\@barredlaction{}%
1441 \def\@eszettaction{}%
1442 \fi
1443 }

```

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

```

1444 \newcommand\@forcecapson{%
1445 \def\AlphaTreatment{2}%
1446 \def\AlphaCapsTreatment{1}%
1447 }

```


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

```

1448 \newcommand\@relaxcapson{%
1449   \let\AlphaTreatment\SaveAlphaTreatment%
1450   \let\AlphaCapsTreatment\SaveAlphaCapsTreatment%
1451   \@defineactions%
1452 }

```

`\@decodepointer` As pertains to arguments 3 and 4 of `\substring`, this routine implements use of the `$` character to mean END-OF-STRING, and `$-integer` for addressing relative to the END-OF-STRING.

```

1453 \newcommand\@decodepointer[2][\value{@stringsize}]{%
1454   \isnextbyte[q]{$}{#2}%
1455   \if T\theresult%
1456     \isnextbyte[q]{-}{\expandafter\@gobble#2}%
1457     \if T\theresult%
1458       \setcounter{@@@letterindex}{#1}%
1459       \@gobblearg{#2}{2}%
1460       \addtocounter{@@@letterindex}{-\gobbledword}%
1461       \edef\@fromtoindex{\value{@@@letterindex}}%
1462     \else
1463       \edef\@fromtoindex{#1}%
1464     \fi
1465   \else
1466     \edef\@fromtoindex{#2}%
1467   \fi
1468 }

```

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

```

1469 \newcommand\@getstringlength[2]{%
1470   \edef\@@teststring{#1\endofstring}%
1471   \ifthenelse{equal{\@@teststring}{\endofstring}}%
1472   {\setcounter{#2}{0}}%
1473   {%
1474     \setcounter{@gobblesize}{1}%
1475     \whiledo{\value{@gobblesize} < \@MAXSTRINGSIZE}{%
1476       %
1477       \@gobblearg{\@@teststring}{1}%
1478       \edef\@@teststring{\gobbledword}%
1479       \ifthenelse{equal{\@@teststring}{\endofstring}}%
1480       {\setcounter{#2}{\value{@gobblesize}}%
1481        \setcounter{@gobblesize}{\@MAXSTRINGSIZE}}%
1482       {\addtocounter{@gobblesize}{1}}%
1483     }%
1484   }%
1485 }

```

`\@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.

```

1486 \newcommand\@gobblearg[2]{%
1487   \setcounter{@letterindex}{0}%
1488   \setcounter{@gobbleindex}{#2}%
1489   \edef\gobbledword{#1}%
1490   \whiledo{\value{@letterindex} < \value{@gobbleindex}}{%
1491     \isnextbyte[q]{\EscapeChar}{\gobbledword}%
1492     \if T\theresult%
1493 %       GOBBLE ESCAPE CHARACTER
1494       \edef\gobbledword{\@DiscardNextChar{\gobbledword}}%
1495     \fi
1496 % GOBBLE NORMAL CHARACTER OR ESCAPE CODE
1497     \edef\gobbledword{\@DiscardNextChar{\gobbledword}}%
1498     \addtocounter{@letterindex}{1}%
1499   }%
1500 }
```

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

```

1501 \newcommand\@DiscardNextChar[1]{%
1502   \expandafter\if\expandafter\BlankSpace#1\else
1503     \expandafter\@gobble#1%
1504   \fi
1505 }
```

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

```

1506 \newcommand\@convertsymboltostring[4][v]{%
1507   \def\@fromcode{#3}%
1508   \def\@tostring{#4}%
1509   \def\EncodingTreatment{e}%
1510   \substring[e]{#2}{1}{\@MAXSTRINGSIZE}%
1511   \@convertoff%
1512   \if e#1\else\substring[#1]{\thestring}{1}{\@MAXSTRINGSIZE}\fi%
1513 }
```

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

```

1514 \newcommand\@convertbytetostring[4][v]{%
1515   \def\@frombyte{#3}%
1516   \def\@tostring{#4}%
1517   \def\EncodingTreatment{e}%
1518   \substring[e]{#2}{1}{\@MAXSTRINGSIZE}%
1519   \@convertoff%
```

```

1520 \if e#1\else\substring[#1]{\thestring}{1}{\@MAXSTRINGSIZE}\fi%
1521 }

```

\@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.

```

1522 \newcommand\@treatleadingspaces[3][v]{\+%
1523 \defaultTreatments%
1524 \edef\thestring{#2}%
1525 \@getstringlength{\thestring}{\@stringsize}%
1526 \setcounter{@maxrotation}{\value{@stringsize}}%
1527 \setcounter{@letterindex}{0}%
1528 \whiledo{\value{@letterindex} < \value{@maxrotation}}{%
1529 \addtocounter{@letterindex}{1}%
1530 \isnextbyte[q]{\EncodedBlankSpace}{\thestring}%
1531 \if F\theresult\isnextbyte[q]{\BlankSpace}{\thestring}\fi%
1532 \if T\theresult%
1533 \isnextbyte[q]{#3}{x}%
1534 \if F\theresult%
1535 % NORMAL OR ENCODED BLANK... ROTATE IT
1536 \edef\thestring{\@rotate{\thestring}}%
1537 \else
1538 % NORMAL OR ENCODED BLANK... DELETE IT (IF 3rd ARG=X)
1539 \@gobblearg{\thestring}{1}%
1540 \edef\thestring{\gobbledword}%
1541 \fi
1542 \else
1543 \setcounter{@maxrotation}{\value{@letterindex}}%
1544 \fi
1545 }~?%
1546 \substring[#1]{\thestring}{1}{\@MAXSTRINGSIZE}%
1547 }

```

\@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.

```

1548 \newcommand\@convertoff{\def\@frombyte{xy}\def\@tostring{}%
1549 \def\@fromcode{xy}}
1550 \@convertoff

```

\@rotate The following code is the engine of the string manipulation routine. It is a tree of successive L^AT_EX 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, the leading character is either encoded, decoded, or removed in the process. Note:

`\@rotate` loses track of double/multi-spaces, per L^AT_EX convention, unless encoded blanks (~) are used.

```

1551 \newcommand\@rotate[1]{%
1552 % CHECK BYTE CONVERSION TEST FIRST
1553 \if \@frombyte#1\@tostring\else
1554 % MUST CHECK FOR MULTI-BYTE CHARACTERS NEXT, SO THAT ENCODING CHARACTER
1555 % ISN'T MISTAKEN FOR A NORMAL CHARACTER LATER IN MACRO.
1556 \if 0\SymbolTreatment%
1557 \@removeExpandableSymbols{#1}%
1558 \else
1559 \@rotateExpandableSymbols{#1}%
1560 \fi
1561 \fi
1562 }
1563
1564 \newcommand\@rotateExpandableSymbols[1]{%
1565 % INDIRECT (EXPANDABLE) SYMBOLS
1566 \expandafter\ifx\expandafter\&#1\&\else
1567 \expandafter\ifx\expandafter\%#1%\else
1568 \expandafter\ifx\expandafter\##1#\else
1569 \@rotateBlankSpaces{#1}%
1570 \fi
1571 \fi
1572 \fi
1573 }
1574
1575 \newcommand\@removeExpandableSymbols[1]{%
1576 % INDIRECT (EXPANDABLE) SYMBOLS
1577 \expandafter\ifx\expandafter\&#1\else
1578 \expandafter\ifx\expandafter\%#1\else
1579 \expandafter\ifx\expandafter\##1\else
1580 \@rotateBlankSpaces{#1}%
1581 \fi
1582 \fi
1583 \fi
1584 }
1585
1586 \newcommand\@rotateBlankSpaces[1]{%
1587 \expandafter\ifx\expandafter$#1$\else% <---RETAIN GOING INTO/FROM MATH MODE
1588 % THE FOLLOWING FINDS TILDES, BUT MUST COME AFTER EXPANDABLE SYMBOL
1589 % SEARCH, OR ELSE IT FINDS THEM TOO, BY MISTAKE.
1590 \if \EncodedBlankSpace#1\@blankaction\else% <--- FINDS REENCODED TILDE
1591 % THE FOLLOWING SHOULD FIND TILDES, BUT DOESN'T... THUS, COMMENTED OUT.
1592 % \expandafter\ifx\expandafter\EncodedBlankSpace#1\@blankaction\else
1593 \if \BlankSpace#1\@blankaction\else
1594 \if 2\AlphaTreatment%
1595 \@chcaseAlpha{#1}%
1596 \else
1597 \if 0\AlphaTreatment%
```

```

1598         \@removeAlpha{#1}%
1599     \else
1600         \@rotateAlpha{#1}%
1601     \fi
1602 \fi
1603 \fi
1604 % \fi
1605 \fi
1606 \fi
1607 }
1608
1609 \newcommand\@rotateAlpha[1]{%
1610 % LOWERCASE
1611 \if a#1a\else
1612 \if b#1b\else
1613 \if c#1c\else
1614 \if d#1d\else
1615 \if e#1e\else
1616 \if f#1f\else
1617 \if g#1g\else
1618 \if h#1h\else
1619 \if i#1i\else
1620 \if j#1j\else
1621 \if k#1k\else
1622 \if l#1l\else
1623 \if m#1m\else
1624 \if n#1n\else
1625 \if o#1o\else
1626 \if p#1p\else
1627 \if q#1q\else
1628 \if r#1r\else
1629 \if s#1s\else
1630 \if t#1t\else
1631 \if u#1u\else
1632 \if v#1v\else
1633 \if w#1w\else
1634 \if x#1x\else
1635 \if y#1y\else
1636 \if z#1z\else
1637 \if 2\AlphaCapsTreatment%
1638     \@chcaseAlphaCaps{#1}%
1639 \else
1640     \if 0\AlphaCapsTreatment%
1641         \@removeAlphaCaps{#1}%
1642     \else
1643         \@rotateAlphaCaps{#1}%
1644     \fi
1645 \fi
1646 \fi
1647 \fi

```

```

1648                                     \fi
1649                                     \fi
1650                                 \fi
1651                                 \fi
1652                                 \fi
1653                                 \fi
1654                                 \fi
1655                                 \fi
1656                                 \fi
1657                                 \fi
1658                                 \fi
1659                                 \fi
1660                                 \fi
1661                                 \fi
1662                                 \fi
1663                                 \fi
1664                                 \fi
1665                                 \fi
1666                                 \fi
1667                                 \fi
1668                                 \fi
1669                                 \fi
1670                                 \fi
1671                                 \fi
1672 }
1673
1674 \newcommand\@removeAlpha[1]{%
1675 % LOWERCASE
1676 \if a#1\else
1677 \if b#1\else
1678 \if c#1\else
1679 \if d#1\else
1680 \if e#1\else
1681 \if f#1\else
1682 \if g#1\else
1683 \if h#1\else
1684 \if i#1\else
1685 \if j#1\else
1686 \if k#1\else
1687 \if l#1\else
1688 \if m#1\else
1689 \if n#1\else
1690 \if o#1\else
1691 \if p#1\else
1692 \if q#1\else
1693 \if r#1\else
1694 \if s#1\else
1695 \if t#1\else
1696 \if u#1\else
1697 \if v#1\else

```

```

1698         \if w#1\else
1699         \if x#1\else
1700         \if y#1\else
1701         \if z#1\else
1702         \if 2\AlphaCapsTreatment%
1703         \@chcaseAlphaCaps{#1}%
1704         \else
1705         \if 0\AlphaCapsTreatment%
1706         \@removeAlphaCaps{#1}%
1707         \else
1708         \@rotateAlphaCaps{#1}%
1709         \fi
1710       \fi
1711     \fi
1712   \fi
1713 \fi
1714 \fi
1715 \fi
1716 \fi
1717 \fi
1718 \fi
1719 \fi
1720 \fi
1721 \fi
1722 \fi
1723 \fi
1724 \fi
1725 \fi
1726 \fi
1727 \fi
1728 \fi
1729 \fi
1730 \fi
1731 \fi
1732 \fi
1733 \fi
1734 \fi
1735 \fi
1736 \fi
1737 }
1738
1739 \newcommand\@chcaseAlpha[1]{%
1740 % LOWERCASE TO UPPERCASE
1741   \if a#1A\else
1742   \if b#1B\else
1743   \if c#1C\else
1744   \if d#1D\else
1745   \if e#1E\else
1746   \if f#1F\else
1747   \if g#1G\else

```

```

1748     \if h#1H\else
1749     \if i#1I\else
1750     \if j#1J\else
1751     \if k#1K\else
1752     \if l#1L\else
1753     \if m#1M\else
1754     \if n#1N\else
1755     \if o#1O\else
1756     \if p#1P\else
1757     \if q#1Q\else
1758     \if r#1R\else
1759     \if s#1S\else
1760     \if t#1T\else
1761     \if u#1U\else
1762     \if v#1V\else
1763     \if w#1W\else
1764     \if x#1X\else
1765     \if y#1Y\else
1766     \if z#1Z\else
1767         \if 2\AlphaCapsTreatment%
1768             \@chcaseAlphaCaps{#1}%
1769         \else
1770             \if 0\AlphaCapsTreatment%
1771                 \@removeAlphaCaps{#1}%
1772             \else
1773                 \@rotateAlphaCaps{#1}%
1774             \fi
1775         \fi
1776     \fi
1777 \fi
1778 \fi
1779 \fi
1780 \fi
1781 \fi
1782 \fi
1783 \fi
1784 \fi
1785 \fi
1786 \fi
1787 \fi
1788 \fi
1789 \fi
1790 \fi
1791 \fi
1792 \fi
1793 \fi
1794 \fi
1795 \fi
1796 \fi
1797 \fi

```



```

1798     \fi
1799     \fi
1800     \fi
1801     \fi
1802 }
1803
1804 \newcommand\@rotateAlphaCaps[1]{%
1805 % UPPERCASE
1806   \if A#1A\else
1807     \if B#1B\else
1808       \if C#1C\else
1809         \if D#1D\else
1810           \if E#1E\else
1811             \if F#1F\else
1812               \if G#1G\else
1813                 \if H#1H\else
1814                   \if I#1I\else
1815                     \if J#1J\else
1816                       \if K#1K\else
1817                         \if L#1L\else
1818                           \if M#1M\else
1819                             \if N#1N\else
1820                               \if O#1O\else
1821                                 \if P#1P\else
1822                                   \if Q#1Q\else
1823                                     \if R#1R\else
1824                                       \if S#1S\else
1825                                         \if T#1T\else
1826                                           \if U#1U\else
1827                                             \if V#1V\else
1828                                               \if W#1W\else
1829                                                 \if X#1X\else
1830                                                   \if Y#1Y\else
1831                                                     \if Z#1Z\else
1832                                                       \if O\NumeralTreatment%
1833                                                         \@removeNumerals{#1}%
1834                                                         \else
1835                                                           \@rotateNumerals{#1}%
1836                                                         \fi
1837                                                       \fi
1838                                                     \fi
1839                                                   \fi
1840                                                 \fi
1841                                               \fi
1842                                             \fi
1843                                           \fi
1844                                         \fi
1845                                       \fi
1846                                     \fi
1847                                   \fi

```

```

1848         \fi
1849     \fi
1850 \fi
1851 \fi
1852 \fi
1853 \fi
1854 \fi
1855 \fi
1856 \fi
1857 \fi
1858 \fi
1859 \fi
1860 \fi
1861 \fi
1862 \fi
1863 }
1864
1865 \newcommand\@removeAlphaCaps[1]{%
1866 % UPPERCASE
1867 \if A#1\else
1868 \if B#1\else
1869 \if C#1\else
1870 \if D#1\else
1871 \if E#1\else
1872 \if F#1\else
1873 \if G#1\else
1874 \if H#1\else
1875 \if I#1\else
1876 \if J#1\else
1877 \if K#1\else
1878 \if L#1\else
1879 \if M#1\else
1880 \if N#1\else
1881 \if O#1\else
1882 \if P#1\else
1883 \if Q#1\else
1884 \if R#1\else
1885 \if S#1\else
1886 \if T#1\else
1887 \if U#1\else
1888 \if V#1\else
1889 \if W#1\else
1890 \if X#1\else
1891 \if Y#1\else
1892 \if Z#1\else
1893 \if O\NumeralTreatment%
1894 \@removeNumerals{#1}%
1895 \else
1896 \@rotateNumerals{#1}%
1897 \fi

```

```

1898                                     \fi
1899                                 \fi
1900                             \fi
1901                         \fi
1902                     \fi
1903                 \fi
1904             \fi
1905         \fi
1906     \fi
1907 \fi
1908 \fi
1909 \fi
1910 \fi
1911 \fi
1912 \fi
1913 \fi
1914 \fi
1915 \fi
1916 \fi
1917 \fi
1918 \fi
1919 \fi
1920 \fi
1921 \fi
1922 \fi
1923 \fi
1924 }
1925
1926 \newcommand\@chcaseAlphaCaps[1]{%
1927 % UPPERCASE TO LOWERCASE
1928 \if A#1a\else
1929 \if B#1b\else
1930 \if C#1c\else
1931 \if D#1d\else
1932 \if E#1e\else
1933 \if F#1f\else
1934 \if G#1g\else
1935 \if H#1h\else
1936 \if I#1i\else
1937 \if J#1j\else
1938 \if K#1k\else
1939 \if L#1l\else
1940 \if M#1m\else
1941 \if N#1n\else
1942 \if O#1o\else
1943 \if P#1p\else
1944 \if Q#1q\else
1945 \if R#1r\else
1946 \if S#1s\else
1947 \if T#1t\else

```

```

1948         \if U#1u\else
1949         \if V#1v\else
1950         \if W#1w\else
1951         \if X#1x\else
1952         \if Y#1y\else
1953         \if Z#1z\else
1954         \if O\NumeralTreatment%
1955         \@removeNumerals{#1}%
1956         \else
1957         \@rotateNumerals{#1}%
1958         \fi
1959         \fi
1960         \fi
1961         \fi
1962         \fi
1963         \fi
1964         \fi
1965         \fi
1966         \fi
1967         \fi
1968         \fi
1969         \fi
1970         \fi
1971         \fi
1972         \fi
1973         \fi
1974         \fi
1975         \fi
1976         \fi
1977         \fi
1978         \fi
1979         \fi
1980         \fi
1981         \fi
1982         \fi
1983         \fi
1984         \fi
1985 }
1986
1987 \newcommand\@rotateNumerals[1]{%
1988 % NUMERALS
1989 \if 1#11\else
1990 \if 2#12\else
1991 \if 3#13\else
1992 \if 4#14\else
1993 \if 5#15\else
1994 \if 6#16\else
1995 \if 7#17\else
1996 \if 8#18\else
1997 \if 9#19\else

```

```

1998         \if 0#10\else
1999         \if 0\PunctuationTreatment%
2000         \@removePunctuation{#1}%
2001         \else
2002         \@rotatePunctuation{#1}%
2003         \fi
2004         \fi
2005         \fi
2006         \fi
2007         \fi
2008         \fi
2009         \fi
2010         \fi
2011         \fi
2012         \fi
2013         \fi
2014 }
2015
2016 \newcommand\@removeNumerals[1]{%
2017 % NUMERALS
2018 \if 1#1\else
2019 \if 2#1\else
2020 \if 3#1\else
2021 \if 4#1\else
2022 \if 5#1\else
2023 \if 6#1\else
2024 \if 7#1\else
2025 \if 8#1\else
2026 \if 9#1\else
2027 \if 0#1\else
2028 \if 0\PunctuationTreatment%
2029 \@removePunctuation{#1}%
2030 \else
2031 \@rotatePunctuation{#1}%
2032 \fi
2033 \fi
2034 \fi
2035 \fi
2036 \fi
2037 \fi
2038 \fi
2039 \fi
2040 \fi
2041 \fi
2042 \fi
2043 }
2044
2045 \newcommand\@rotatePunctuation[1]{%
2046 % PUNCTUATION
2047 \if ;#1;\else

```

```

2048 \if :#1:\else
2049 \if '#1'\else
2050 \if "#1"\else
2051 \if ,#1,\else
2052 \if .#1.\else
2053 \if ?#1?\else
2054 \if '#1'\else
2055 \if !#1!\else
2056 \if 0\SymbolTreatment%
2057 \@removeDirectSymbols{#1}%
2058 \else
2059 \@rotateDirectSymbols{#1}%
2060 \fi
2061 \fi
2062 \fi
2063 \fi
2064 \fi
2065 \fi
2066 \fi
2067 \fi
2068 \fi
2069 \fi
2070 }
2071
2072 \newcommand\@removePunctuation[1]{%
2073 % PUNCTUATION
2074 \if ;#1\else
2075 \if :#1\else
2076 \if '#1\else
2077 \if "#1\else
2078 \if ,#1\else
2079 \if .#1\else
2080 \if ?#1\else
2081 \if '#1\else
2082 \if !#1\else
2083 \if 0\SymbolTreatment%
2084 \@removeDirectSymbols{#1}%
2085 \else
2086 \@rotateDirectSymbols{#1}%
2087 \fi
2088 \fi
2089 \fi
2090 \fi
2091 \fi
2092 \fi
2093 \fi
2094 \fi
2095 \fi
2096 \fi
2097 }

```

```

2098
2099 \newcommand\@rotateDirectSymbols[1]{%
2100 % DIRECT SYMBOLS
2101 \if /#1/\else
2102 \if @#1@\else
2103 \if *#1*\else
2104 \if (#1(\else
2105 \if )#1)\else
2106 \if -#1-\else
2107 \if _#1_\else
2108 \if =#1=\else
2109 \if +#1+\else
2110 \if [#1[\else
2111 \if ]#1]\else
2112 \if ^#1^\else% <--FOR SUPERSSCRIPTS, NOT \^
2113 \if <#1<\else
2114 \if >#1>\else
2115 \if |#1|\else
2116 \if &#1&\else
2117 \@rotateUndecipherable{#1}%
2118 \fi
2119 \fi
2120 \fi
2121 \fi
2122 \fi
2123 \fi
2124 \fi
2125 \fi
2126 \fi
2127 \fi
2128 \fi
2129 \fi
2130 \fi
2131 \fi
2132 \fi
2133 \fi
2134 }
2135
2136 \newcommand\@removeDirectSymbols[1]{%
2137 % DIRECT SYMBOLS
2138 \if /#1\else
2139 \if @#1\else
2140 \if *#1\else
2141 \if (#1\else
2142 \if )#1\else
2143 \if -#1\else
2144 \if _#1\else
2145 \if =#1\else
2146 \if +#1\else
2147 \if [#1\else

```

```

2148         \if ]#1\else
2149         \if ^#1\else% <---FOR SUPERSCRIPTS, NOT \^
2150         \if <#1\else
2151         \if >#1\else
2152         \if |#1\else
2153         \if &#1\else
2154         \@rotateUndecipherable{#1}%
2155         \fi
2156         \fi
2157         \fi
2158         \fi
2159         \fi
2160         \fi
2161         \fi
2162         \fi
2163         \fi
2164         \fi
2165         \fi
2166         \fi
2167         \fi
2168         \fi
2169         \fi
2170         \fi
2171 }
2172
2173 \newcommand\@rotateUndecipherable[1]{%
2174 % REPLACE UNDECIPHERABLE SYMBOL WITH A TOKEN CHARACTER (DEFAULT .)
2175 \expandafter\@gobble#1\undecipherable%
2176 % DONE... CLOSE UP SHOP
2177 }

2178 \catcode'\&=4
2179 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2180 \</package>

```