tersen

Release 0.1.0

Soren Bjornstad

Dec 16, 2020

CONTENTS

Tersen is a *fast, flexible abbreviation engine* that compresses text in a human-readable fashion. Abbreviations are entirely user-specifiable through a dictionary of textual mappings (e.g., `and` becomes `&`). More concise dictionary files and custom abbreviation behavior can be obtained by writing Lua functions called *annotations* (which pre-process lines in the abbreviation dictionary) and *hooks* (which alter tersen's behavior as it abbreviates a text).

Use cases for tersen include:

- Packing more information onto a cheat sheet or reference guide.
- Sending content over SMS or another limited-bandwidth communication channel.
- Obfuscating content so others cannot easily read it but you can.
- Practicing your reading skills in your favorite alphabetic shorthand system.

Tersen is written and extended in Lua. It uses the MIT license. You can install it using LuaRocks:

```
$ luarocks install tersen
```

CHAPTER ONE

QUICK START

Want to get your hands dirty with tersen? You've come to the right place!

1. If you have not done so already, install Lua and LuaRocks, then run `luarocks install tersen` to install tersen.
2. Copy the following to a new text file and save it as `tersen_dict.txt`:

   ```
   and => &
   with => w/
   without => w/o
   Personal Identification Number => PIN @n_acro
   ```

   This is a *dictionary*, which explains how words are abbreviated in your preferred abbreviation system.

   Each line is called a *mapping*. The part of a mapping before the => is called a *source*, and is what tersen will try to replace in your input file. The part after the => and before the `@` or line break is called a *destination*, and is what tersen will replace the source with when it encounters the source in your input file.

   The bit starting with an `@` is an *annotation*; that causes tersen to add a plural form of the acronym, so that it knows how to abbreviate `Personal Identification Numbers` to `PINs` as well.
3. Create a test input file containing one or more of the source phrases above and save it as `test.txt`.
4. Run tersen:

   ```
   $ tersen tersen_dict.txt test.txt
   ```

   An abbreviated version of your test file will be printed to the terminal. The first argument to `tersen` is the dictionary file; any number of input files can be specified thereafter, and tersen will read them in turn and print the tersened versions to standard output.

At this point, you are ready to build up a dictionary for your preferred system. Check out *Terms* and *The tersen dictionary format* if you get mixed up about the syntax of the dictionary file or how punctuation is handled. If you are building a complex dictionary, you may also want to read about *annotations* and *hooks* before you get too far in, as they may allow you to simplify your dictionary somewhat.

CHAPTER

TWO

TERMS

2.1 Mappings

To tell tersen how you want to abbreviate things, you create a *tersen dictionary* which contains one or more *mappings*. A mapping describes a correspondence between a *source* (a long word or phrase that may occur in the input text) and a *destination* (the abbreviated version you want to replace that long word or phrase with).

2.2 Tokens

Tokens are the units of input text tersen attempts to match with your dictionary mappings as it decides what to replace. *Outer tokens* are strings of characters separated by whitespace. Each outer token has an associated *inner token*, which is the part of the outer token beginning at the first inner-token character and ending at the first non-inner-token character. Inner-token characters are the alphanumeric characters in your system locale, `-`, `'`, and `’`. The portions of an outer token that are not part of the inner token are the *initial* and *final* portions, or, together, the *token perimeter*.

Only the inner token is used when searching for possible replacements, but the token perimeter is preserved in the output (e.g., with the mapping presented above, if `Internet,` was found in the source, `I.N.,` would appear in the destination).

Note: The definition above allows the final portion of the token to contain alphanumeric characters. This might happen, for instance, if your source contains a phrase like `Wolfram|Alpha` – `Wolfram` would be matched, while `|Alpha` would be ignored and passed through verbatim.

2.3 Sources

A source may consist of one *token* or several tokens. When tersen encounters an input token that could begin multiple mappings, it looks ahead and picks the longest possible match. For instance, if your dictionary contains sources for both `Internet` and `Internet Protocol`, the phrase `the Internet is` becomes `the I.N. is`, while `Internet Protocol` becomes `IP` (never `I.N. Protocol`). The “longest possible match” is the one that consumes the most tokens from the input right now; it is not necessarily the way of dividing tokens that produces the shortest output. (tersen is *greedy*; it will never backtrack over a replacement it has already made, even if another division of tokens could produce a shorter output. While that gives up a small amount of possible compression, it keeps tersen fast and simple and makes replacement behavior more predictable.)

A source may consist of only inner tokens; that is, it cannot contain punctuation other than hyphens or apostrophes. So you cannot have a source of, say, `#&$%`, or `St. Paul`. If such a source is found, a warning will be printed and that mapping will be ignored. However, this does not preclude matching phrases that have punctuation in the middle; if a multi-word phrase matches input when ignoring the punctuation in the input, the replacement will still be made and any medial punctuation will disappear. For instance, if you have a mapping from `St Olaf` to `STO`, and the text `St. Olaf` is found in the input, it will be replaced with `STO`. One could imagine a case where this would do the wrong thing, such as `222 Somewhere St., Olaf City, CA`, but in general this is unlikely (and it will always be possible to fool tersen in some edge cases, because language is complicated!).

Only one mapping may be present in the dictionary for each source. If the same source is mapped more than once, the mapping that comes physically first in the source file wins. For mappings beyond the first, tersen will print a warning, unless the duplicate is programmically generated by an annotation or the `-` flag is used on the entry (see later for more on annotations and flags). This behavior can be customized via the `mapping_conflicts` *hook*, so you can have later mappings overwrite earlier ones, for example.

2.4 Destinations

A destination may be any string containing any characters, including whitespace and punctuation, with the sole exception of the newline and the at-sign (`@`), which both indicate the end of the destination string (the at-sign additionally applies an *annotation*). Leading and trailing whitespace in the destination string are ignored.

If a destination is longer (consists of more characters) than its corresponding source, tersen will print a warning but still use the mapping. This behavior can be customized via the `mapping_verbosens_text` hook.

CHAPTER

THREE

THE TERSEN DICTIONARY FORMAT

If you have not yet read *Terms*, you may wish to do so before reading this section, as it will freely use the terms presented there.

You can find a full-featured example tersen dictionary based on Dutton Speedwords in the `examples/` directory of the source distribution; note that it relies on the example annotations file to function correctly.

3.1 Basics

The abbreviation table is defined in a *tersen dictionary*, a text file with the following format.

Lines beginning with the comment character `#` and blank lines are ignored. All other lines create one or more mappings. The ordering of entries in the dictionary does not matter, unless you map the same source to multiple destinations, in which case the mapping listed first in the file wins and the rest are ignored (but a warning is printed to let you know there might be a mistake, unless you explicitly *suppress it*).

The simplest kind of dictionary line looks like this:

```
source => destination
```

This creates a mapping that will replace `source` with `destination` in output.

You may want to map several sources to the same destination. You can do this by separating them with commas:

```
source1, source2 => destination
```

3.2 Annotated mappings

You can apply an *annotation* to any dictionary line by putting an at-sign, the name of the annotation, and optionally some parameters at the end of the line. Annotations programmatically post-process your mapping in some way, reducing the number of repetitive entries that you need to include in your dictionary file.

There are three different forms of annotations. The first is *argumentless*:

```
tree => bo @n
```

The second uses a single pair of square brackets to delimit the arguments, and individual arguments are separated by spaces:

```
easy => fas @adj[easier easiest]
```

Each argument can contain any character except whitespace and closing square brackets.

The third and last uses curly braces, with each argument in a separate pair of braces:

```
log in => lgn @v{logs in}{logged in}{logged in}{logging in}
```

Each argument can contain any character except newlines and closing curly braces.

For instance, you might write:

```
you're => v_e @apos
```

Or:

```
easy => fas @adj[easier easiest]
```

If you annotate a line that has comma-separated sources, the annotation function will be applied to each of the sources in turn.

See the *Annotations* section for details on tersen's built-in annotations and how you can write your own.

3.3 Flags

The following characters of punctuation, called *flags*, when placed at the start of a line in the dictionary, have special effects. Any number of flags can be used together, and if the same flag is used twice, tersen will behave as if it were used only once.

! The *cut* flag causes tersen to stop parsing the dictionary immediately after this entry. This may be useful if you're trying to debug a small portion of the dictionary. To reduce the risk of accidentally leaving a cut in the dictionary file, a warning will be printed anytime a cut is present, indicating which line the cut is on.

? The *trace* flag causes tersen to print its internal lookup-table structure for all entries generated by this dictionary line. This can be useful when debugging annotations. For multi-word tokens, the structure will be printed back from the first token (so if "Internet Protocol" has a ? by it, the entry for "Internet" will appear, containing "Internet Protocol" as a continuation member).

A warning is printed anytime a trace is present.

Note: The trace is printed for all flagged items only after the entire lookup table is built. Therefore, if the entry being traced is not actually inserted successfully (for instance, because it had the same source as an earlier entry), it won't show up in a trace.

– The *suppress redefinition* flag silences any warnings that would otherwise be displayed if any mappings created by this line conflict with existing mappings. This only affects the display of the warning; the original mappings will win, as they would without the flag. This flag is useful when using an annotation that happens to generate an item with the same source as a previous mapping; for instance, the present tense of the verb *lead* and the singular form of the noun *lead* are identical, but you might want to include their base forms and attach noun and verb annotations to them in the dictionary.

CHAPTER

FOUR

ANNOTATIONS

Annotations are Lua functions that transform the source and destination of a mapping in an arbitrary way. An annotation can output one mapping or many mappings; for instance, you might wish to add both the singular form and a plural form of a word using an annotation. Several annotations are provided with tersen, and you can easily write your own.

4.1 Dictionary syntax

The details of how annotations are written in the dictionary can be found at *Annotated mappings*.

4.2 Built-in annotations

The following annotations are included with tersen.

n_acro Apply to an acronym to allow tersen to recognize its English plural form as well. For example:

```
Personal Identification Number => PIN @n_acro
```

This will add an extra entry mapping `Personal Identification Numbers` to `PINs`.

If the plural were not `Personal Identification Numbers`, but, say, `People Identification Number`, you could list the source's plural form as an argument:

```
Personal Identification Number => PIN @n_acro{People Identification Number}
```

apos Apply to a word containing an apostrophe so that it works with both curly and straight apostrophes. For example:

```
you're => v_e @apos
```

This will add entries mapping both `you're` and `you’re` to `v_e`.

numbers Ignore the provided source and destination and instead add entries mapping the English numbers from `one` to `ninety-nine` to the Arabic numerals from `1` to `99`.

Attention: If you specify a custom annotations file with the `-a` option to tersen, you need to include these default annotations in that custom annotations file or they will not be accessible. See *Backing functions*, below, for more details on creating such a file.

4.3 Backing functions

Annotations are processed by Lua functions in the `M` table in the `tersen/extend/annot.lua` file in the tersen distribution. The name of the function is the name used to access the annotation in the tersen dictionary, and may contain lowercase letters, numbers, and underscores (annotations in the tersen dictionary are flattened to lowercase, so if you use uppercase in your name it will be impossible to call it).

If you want to add your own annotations, you should start from a copy of this file, putting it somewhere convenient and telling tersen to use it with the `-a annotationfile` argument at runtime. If you don't want to type this every time you run tersen, *set up* a `.tersenrc`.

4.4 Function details

Each annotation function takes three arguments: the source listed in the tersen dictionary, the destination listed in the tersen dictionary, and a list of arguments to the annotation (as a table with numeric keys, or `nil` if the annotation was argumentless). It returns a table of mappings, the keys being sources and the values being destinations.

If there are multiple comma-separated sources, the annotation function is called separately for each source.

Here's a simple example, the built-in function for `@apos`, which ensures that any apostrophes in the source value get dictionary entries for both curly apostrophes and straight apostrophes.

```
function M.apos (source, dest, args)
    local straight = string.gsub(source, "'", "’")
    local curly = string.gsub(source, "’", "'")
    return {[straight] = dest, [curly] = dest}
end
```

It's possible to use an annotation to programmatically generate some entries without using the mapping at all. For instance, the built-in `@numbers` annotation adds entries for the words "one" through "ninety-nine" and their corresponding digit representations. To include the output of such an annotation in your dictionary, simply put a dummy source and replacement on a line and attach the annotation, like so:

```
Numbers as Words => Digits @numbers
```

You can trace this line to see the effect if you like, by placing a *?* in front of it (see *Flags*, below).

If you find this to be an ugly abuse of annotations, you can also get the programmatic-generation effect using the `post_build_lut` *hook*.

CHAPTER

FIVE

HOOKS

Tersen is intended to have sane default behavior for most use cases, but there are a wide variety of languages and abbreviation systems in the world, so its default behavior may not be suitable for all of them. To add some flexibility, tersen has various *hooks* scattered throughout its codebase.

By uncommenting a hook and giving it an implementation in `hooks.lua`, you can change the behavior of a small piece of tersen's process. For instance, you might want to change what happens if a source *is defined twice in the dictionary*, or *how the case of replacements is determined*. The names of hooks that can be used to customize behavior are mentioned throughout this manual. If you find there is no hook present for an aspect of behavior you need to change, pull requests adding such a hook are welcome.

To get started, grab a copy of the `hooks.lua`, which has commented-out stub implementations and extensive comments on all the available hooks. When you run tersen, supply the `-h hookfile` argument. If you don't want to type this every time you run tersen, *set up* a `.tersenrc`.

CHAPTER

SIX

COMMAND-LINE OPTIONS

You can find a full accounting of tersen's command-line options by running `tersen --help`.

6.1 tersenrc

On startup, tersen looks for a file called `.tersenrc` in your home directory. If found, each line in the file will be treated as if it was passed as a command-line argument to tersen, with all arguments in the `.tersenrc` coming before any that you actually type on the command line.

If you don't want to use the options in your `.tersenrc` for a given run, pass the `--no-rcfile` option.

CHAPTER

SEVEN

IMPLEMENTATION DETAILS

Most users will not care about these details because tersen will just do the right thing. However, if you're encountering odd behavior or you wish to extend tersen, you might want to have a look at these notes.

7.1 Case

Tersen is case-insensitive when matching, to the extent that your system locale can lowercase the characters you're working with. When replacing a match, tersen uses some simple rules to determine the case of the output:

- If the destination, as defined in the tersen dictionary, is all-caps, tersen assumes this is an acronym, and the output will always be all-caps.
- If the matched text itself was all-caps, its tersened output will be as well.
- If the matched text was title case, its tersened output will be in title case.
- Otherwise, the output will be whatever case the replacement is.

Though these rules are not always perfect, for the most part, tersen will do the right thing with case. If you find it is doing the wrong thing, you can customize the rules via the `normalize_case` hook.

7.2 Unicode

Lua is 8-bit clean and works great with Unicode inputs and dictionary files. For best results, you should ensure your system locale is set appropriately, usually to UTF-8 (otherwise Lua may have the wrong idea of what constitutes an alphanumeric character, for instance).

However, tersen does not attempt to *normalize* Unicode, which means that it may occasionally miss possible matches if a dictionary source and the input tokens are written in different ways (for instance, one uses combining characters and the other does not). If this is a problem for your use case, you should use another utility such as uconv to normalize your dictionary file and inputs prior to invoking tersen.

7.3 Dot-coalescing

If a replacement ends with `.` but a `.` already comes after the matched tokens in the source, tersen will remove exactly one dot from the sequence; this prevents sentences ending with `..` because the last word was substituted with a period-ending abbreviation. (If the matched tokens had *more* than one dot after them – for instance, if the sentence ended in an ellipsis – only one of them will be removed.)

7.4 Performance

tersen is designed to be fast enough that performance should not normally be a concern. However, depending on what features you take advantage of, what kind of speed you require, and how much input you intend to pass to tersen, you may want to consider a few things.

Building the lookup table is almost instantaneous in most cases, even with annotations. A 700-line tersen dictionary with liberal use of moderately complex annotations and multiple sources takes under 10ms to build on my computer (NB: my development machine is unusually fast for a desktop). Thus, the main factor is usually how large your input is; on the same machine and a large corpus, tersen works at about 0.45 seconds per megabyte of input. For comparison, a megabyte is about 180,000 words of English text, so tersen processes about 400,000 words per second in this benchmark.

A significant secondary factor is the use of hooks, particularly `no_match` and `normalize_case`. Since these functions run against a large proportion of the words in the source text, these are very "hot" functions and almost anything you can do to speed them up will likely have a noticeable impact if tersen is running slowly for you.