ChatScript Spelling Marking Manual

Copyright Bruce Wilcox, mailto:gowilcox@gmail.com www.brilligunderstanding.com
Revision 3/21/2021 cs11.2 0

Spelling

One (of many) unusual features of ChatScript is the built-in spell-checker. Most spell checkers are built to suggest possible replacements. But ChatScript's spell-checker is completely responsible for deciding what substitution to make and to make it in an environment (chat) where casing of words is highly unreliable.

OK. So it does the best it can with unknown words. It is enabled by default but your scripts can disable it. All you have to do is assign a different value to $cs_token, one which does not include #DO_SPELLCHECK. E.g.

```
$cs_token = #DO_PRIVATE | #NO_SEMICOLON_END  | #DO_SUBSTITUTE_SYSTEM   | #DO_SPELLCHECK  | #
$cs_token -= #DO_SPELLCHECK     # disable spellcheck
```

You can do this dynamically. If you know the user's next input is likely to have issues (you have asked for his name and foreign names will get messed up), you can turn it off, go read in his name unchecked, and then turn it on again.

You can do this after-the-fact. You can tell that the input sentence got spell checked because %tokenFlags will have a bit value of #DO_SPELLCHECK set for it. You can get back the original value of some word position using ˆoriginal. You can also turn off spell checking and then take a copy of the input sentence and do ˆanalyze or ˆretry(INPUT) or some such behavior to get it unspell checked.

Casing

A normal dictionary has one copy of a word and multiple definitions, some perhaps in lower case and some in upper case. The ChatScript dictionary keeps separate entries for a lower case form and multiple upper case forms (e.g., GE the company and Ge the element symbol). This allows ChatScript to keep facts using the correct form of a word.

Casing in Patterns

When you write patterns using words directly, you should write them in the case that the word SHOULD be. E.g.,

```
u: (i love you) --
```

This rule is badly written because the correct case of `i` should be `I`. When the user writes whatever they do, spell correction will likely change the case of the input and then it won't match your rule. The script compiler will issue warning messages when you write rules where the word in a pattern occurs only in some other case in the dictionary. If you are wrong, correct your code. If you are not, leave it alone.

Pay no attention to the issue of words at the start of a sentence being capitalized. That is not how they would be found in a dictionary. CS will change them to lower case during analysis if it feels it is appropriate.

In various circumstances ChatScript may manage a match anyway. But if you use memorization (_) to get what the user said and echo it back to the user, you really would prefer to echo it back in the correct case. Your bot will look smarter.

Casing in Concepts

Correct case is also something you do within concept sets. ChatScript can track whether the word is in lower or upper case in some concept set, so when you have a rule like this:

```
u: (_~animals) I like '_0 too.
```

it won't matter what casing the user uses, the value will be converted to the case used by the concept set.

Debug commands `:dualupper :mixedcase`

ChatScript can help you find badly cased words. One of the debug commands you can use will list all entries in the dictionary that are multiple forms of upper case spellings (:dualupper). You can use that to help correct some of your data. A different debug command (:mixedcase) will list words that occur in both cases and that are not listed in WordNet.

Multiple forms of input

You can imagine that ChatScript makes equally available a number of forms of the input sentence. The first form, %originalInput, is literally what the user typed in (all sentences at once sans OOB). Or you might want %originalSentence, the raw content of the current sentence after ChatScript has tokenized things and decided where the sentence ends.

After tokenization, ChatScript will have available each word of the sentence in both its adjusted form (after all sorts of preprocessing including spell checking), and its canonical form (sometimes called lemma in NL). ChatScript canonical forms are aimed to support understanding meaning, so go beyond mere lemmas. Lemmas normally exist for nouns, verbs, adjectives, and adverbs. ChatScript includes canonical forms for pronouns and numbers. You can use either input stream (adjusted or canonical) during pattern matching and usually you will use both, depending on how you pick your keywords in the pattern.

Yet another form of input is ontology marking. Every word will have associated with it the concepts it is directly or indirectly a member of. You can match those just as easily as matching a word. The concepts a word may be in include statically defined ones (like ~animals) as well as ones resulting from parsing (~mainsubject and ~verb).

Fixing Spelling and ontology marking

ChatScript is never going to make perfect spelling decisions all the time, nor will it be able to perfectly distinguish ontology markings. If someone writes “u” should that mean “you” or is that just a letter. Should it be changed or left alone? Or the phrase “lie to me” might refer to a TV show or might just be ordinary words. It requires context to make that decision, context that the CS engine doesnt have but that your scripts can.

So ChatScript provides you with mechanisms to guide or fix decisions the engine makes. The first fixes are just lists of adjustments to automatically make (data from LIVEDATA/ENGLISH/SUBSTITUTES. Each line has what to see on the left, and what to convert it to on the right. You can put in common spelling errors and their correction, british words and their american spelling, etc. There are lots there already. You enable their use by the value of $CS_TOKEN, which by default has #DO_SUBSTITUTE_SYSTEM (do all files) but which can be broken down to enable and disable specific files.
Furthermore in your scripts you can define new substitutions using `replace:` (but you have to enable #DO_PRIVATE in $CS_TOKEN).

replace:

Just like the LIVEDATA substitutions files, `replace:` takes a pair of things, the first is what you expect to find in user input and the second is what to replace it with. In that first thing, you can detect multiple words (a phrase) in any of 3 ways:

```
replace: "my little pony" xx
replace: my+little+pony xx
replace: my_little_pony xx
```

Internally, all three represent like the third (using underscores). The advantage of using double quotes is a) documentation clarity to show it's a standard recognizable phrase and not just 3 words and b) when your stuff has internal punctuation. You don't know how the tokenizer is going to represent that (usually by separating punctuation into separate tokens with underscores). If use use doublequotes, CS will figure out out correctly for you.

In the second token, if you use double quotes or +, they list separate words.

```
replace: box my+boy
replace: box "my boy"
```

The above replaces 1 word with 2. If you use underscores, they remain a single word so That

```
replace: box my_boy
```

replaces 1 word with 1. If you want to replace literally 1 for 1 with some complex 2nd value (like one that needs + or space in it) when quote a double quoted string.

```
replace: box `"this+box is fun"
```

And even after the sentence is transformed, you can use `^mark` and `^unmark` to mark concepts and words or to turn off marking on them, so you can use context to decide if "lie to me" is a tv show or not, or if you hate 'good_day' being considered ~emogoodbye you can do something like this:

```
u: (_~emogoodbye) if (^original(_0) == good_day) { ^unmark(~emogoodbye _0)}
```

Conditional substitution

If you want to replace something only if some specific word does not follow the match, you can use this format:

```
replace: jack_rusell ![terrier]Jack+Russell+terrier
```

This says to replace jack_russell with Jack+Russel+terrior ONLY if the word immediately following the match is not terrier (case insensitive).

Numeric Substitutions replace: ?_xxx xxxx

?_ on the left takes numbers that have been conjoined with words, and separates them. Input like "I weigh 5kg." is hard to handle via substitition since there are infinite number of numbers. But it's common to see stuff like that from users.

```
replace: ?_gram gram        -- this merely separates
replace: ?_kg kilogram -- this separates into full unit name
replace: ?_kg kilo+gram -- this separates into multiword replacement
```

You should use the singular form of the word. CS can then do singular or plural as needed.

When using full unit name subsitution, you need to be careful that your bot has only 1 expected meaning. "about 1g" – is that 1 gram or 1 gravity? Also it's a good idea to expand to singular forms as that, being canonical, will match more of your patterns.

The ?_xxx notation also works when the unit measure is not attached to the number. Using the first replace above, the input "7 g" will be transformed to "7 gram".

Note that is the number value is 1, you get the singular word. Otherwise you get the plural word. If CS does not know how to make the plural, you can add the pair to LIVEDATA/ENGLISH/plurals.txt

In multiple word substitutions, normally cs pluralizes the last word only. But if the substitution is multiple word 'x per y', CS will pluralize the first word.

Emoji

Emojis are a form of spelling correction. You can replace appropriate utf8 emoji words with spelling replacements that start and end in a colon. These will be considered emoji and marked as ~emoji. E.g., in script you can write

```
replace: xx :palms_up_together:
replace: yy :grinning_face_with_smiling_eyes:
```

where xx and yy are actual utf8 emoji characters.

Or you can add these paired words into one of the LIVEDATA substitution files. Then you can write patterns like: "' u: (:palms_up_together:) Peace be unto you. u: (~emoji) I don't speak emoji.