ChatScript Practicum: Spelling and Interjections

Revision 5/30/2020 cs10.4

"'There's more than one way to skin a cat"'. A problem often has more than one solution. This is certainly true with ChatScript. The purpose of the Practicum series is to show you how to think about features of ChatScript and what guidelines to follow in designing and coding your bot.

Spell checking

Many natural language tools are designed to work with Wall Street Journal sentences - perfectly punctuated and perfectly spelled. That's not the real world of chat. ChatScript deals with the real world by providing a number of mechanisms to correct badly written sentences.

Variant keywords

User input matches patterns of words or concepts. So the first obvious preparation for faulty user input is to list it in your pattern or concept.

```
u: YOULIKE (Do you want *~3 [apple apl aplle appel]) No thanks.
```

Above we are trying to catch bad spellings of apple. Part of the problem here is that some misspellings are so uncommon, it's probably not worth the effort of guessing them in advance (in fact, for many of those you might hope an automatic spell-checker would solve the problem).

Another part of the problem is inconsistency. Elsewhere if you have to recognize `apple`, you may miss some of your choices.

```
u: CANIHAVE (Can I have *~3 [apple  aplle ]) I don't have any.
```

Fixing this problem leads you to moving the list into a concept set.

```
concept: ~apple(apple apl aplle appel)
...
u: YOULIKE (Do you want *~3 ~apple) No thanks.
...
u: CANIHAVE (Can I have *~3 ~apple) I don't have any.
```

This isn't bad. It's clear what is going on. But it still has issues.

One is that you are going to make a lot of concept sets if you are trying to handle deviant spelling, and that costs memory space. Which usually doesn't matter.

Another problem is that you can't just write the misspellings of the word apple. You also have to write the plural misspellings of them, because since they are

not real words, ChatScript cannot conjugate them for you. It doesn't know that `aplle` is a noun.

The more serious problem arises when you want to spit back something the user said. You might want to detect various fruits, so you make a concept ~fruit, which contains ~apple, ~orange, ~banana. Each of those has your deviant spellings. Then you write this rule:

```
?: (do you like _~fruit)  I love '_0.
```

If the user says "do you like aplle", you don't really want your bot to reply "I love aplle". The user may have been sloppy in their input but your bot either looks like a complete idiot, or else it is pointing out the user's mistake and being impolite.

And a final problem is that every keyword you write in a pattern or concept set is added to the dictionary and is considered a legitmate word by CS. Since CS actually has built-in spell-checking, The system may take some other mispelled word and correct it to one of your misspellings.

Partial matching: `*ing bottle* 8bott*`

You can request a match against a partial spelling of an original word (not its canonical form) in various ways. If you use `*` somewhere after an alpha, it matches any number of characters. If you use `*` followed by an alpha, you get anything as a prefix followed by what you request. Put a number in front, it means the word must be exactly that many characters long, matching your pattern. When using an `*` word, you can use `.` to indicate exactly one character of any value.

```
u: ( sag*us )   matches many misspellings of _Sagittarius_.
u: ( *tha )     matches _Martha_.
u: ( 6sit* )    matches _sitter_.
u: ( sit*u.tion )  matches _situation_.
```

This works well enough when there are going to be many possible misspellings, but they will all likely occur later in the word. But of course when used badly it will allow normal words to match wrongly.

You don't really want to use these patterns in multiple places for the same word, and they don't work in concept sets. What you really want is true spelling correction. You want the user's badly typed words to be fixed during the input phase into correctly spelled words, so your patterns and concept sets can be clear and simple. You want input substitution.

Substitutions

ChatScript has files with pairs of words, wherein if the first word is seen in any casing, it is replaced by the second word.

LIVEDATA/ENGLISH/SUBSTITUTES

CS has files for:

```
spellfix - the most commonly misspelled words
british - British spellings converting to American
noise - words and phrases that can be erased from input
texting - expanded form of texting shorthands
contractions - full expansion of them
substitutions - misc.
interjections - convert phrase into interjection/dialog_act
```

The substitutions file may remap multiple words to a single underscored form, or break apart. Or it may just fix more spelling errors. Or convert complex phrases to simple ones, like employment_opportunities ==> jobs

Using these files is enabled (by default) in the $cs_token value in your bot definition macro and you can turn them off.

```
$cs_token = #DO_SUBSTITUTE_SYSTEM
$cs_token -= #DO_TEXTING
```

The syntax of the word pairs is that the badword (1st word) is case insensitive and you can match a phrase either by double-quoting it or by using underscores in it. You can request it match only at beginning of sentence by using < and that it end the sentence using >. Using underscores on the goodword will keep the result as a single word. Using + will split the output into multiple words.

```
dunno I+do+not+know   -- converts word dunno into 4 words
"employment opportunities" jobs  -- simplifies phrase
eluded_to alluded+to   -- fixes false phrase
like_what> for+example -- convert only at end of sentence
<you_have_to you+must -- convert only at start of sentence
```

Script substitutions replace:

Substitutions can be done from script as well, just by putting them after a `replace:` top level script item. I do this a lot to smooth out inputs. I look up a standard way a word is spelled (eg in dictionary or Wikipedia) and make that the true word.

```
replace: ps_2 PlayStation+2
replace: "ps 2" PlayStation+2   -- alternate way to say same thing
replace: bluray Blu-ray
replace: blueray Blu-ray
replace: blue_ray Blu-ray
replace: blu_ray Blu-ray
replace: blueray Blu-ray
replace: bluray Blu-ray
```

WARNING: Substitutions alter the dictionary and end up shared in ALL bots compiled together. They will all perform the substitution.

Numeric Substitutions There is one really cool substitution syntax. It 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: ?_kg kg        -- this merely separates
replace: ?_kg kilograms -- this separates into full unit name
```

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? You may be better off just separating the words and then using context to convert the unit of measure later.

Rule-based substitutions

Prior substitutions all happened as input was being marked in preparation for execute your scripts. But you can also do substitutions while executing your scripts.

```
u: (_bluray) ^replaceword(Blu-ray _0)
```

The above immediately rewrites that word in the sentence. It does NOT change any markings. You can do that using ^mark and ^unmark as you see fit.

You can hide a word or phrase that has been matched by merely by using ^unmark(* _0) upon it. Thereafter it is invisible unless you do a ^mark to revert it.

When you want more radical change, you can do this:

```
s: (_* find me _*)
    $_join = ^join( "never have I seen" '_0 " " '_1' .)
    $_join = ^substitute(character $_join _ " ")
    ^analyze($_join)
        or
    ^input($_join)
    ^fail(SENTENCE)
```

Here we delete words of the current sentence, add new ones (keeping the same punctuation), and then either continue processing after this rule with the new sentence (fully marked) OR stash this as the next sentence and exit our script and come back in with this new sentence, before continuing with other sentences of the user's input. When you can organize your rules to support use of ^analyze and then Just allow rules to continue, that is most efficient (avoiding coming into the control script yet again). The ^substitute call is in case you have underscores instead of blanks being used when memorizing input.

Just to be clear, when you use ^mark, it doesn't change the word, just what is marked about it for pattern matching. You can, for example, do this:

```
u: TEST (_test) ^mark(rhino _0)
u: FINDTEST(_test) I found '_0.
u: RHINO (_rhino)  I matched '_0
```

which finds where "test" is in the sentence and also adds a mark for "rhino". It does not mark logical concept implications of rhino, just the word. Then, give input "what is the test". The FINDTEST rule will find it and print out "I found test". The RHINO rule will match also, and print out "I matched test".

Automatic Spell checking

Ideally you'd like CS to just fix words for you. And often it can do that. It compares the unknown word with the dictionary and finds out how many transformations of letters it would take to get to some other word. Some transformations cost more than others, and if the cost is too high it doesnt not accept that change (after all you can get to ANY word with enough things like add a letter, delete a letter, swap adjacent letters, change a letter)

Spell check will find words it could allow, and then ranks them based on how common the word is, at what age the word is learned, and whether the word has the same start and end (since people usually get that right).

It also has a bunch of heuristics that allow it to break up a word into two (for run-together words) or join two adjacent words, and do other transformations.

Automatic spell checking is why you really don't want to put badly spelled words in your patterns or concepts.

But automatic spell checking can't handle everything, so When you see obvious errors that are common, you add them to your replace: scripts.

It is possible to disable automatic spell checking, but there isn't much incentive. You can readily access what the user actually typed in for any word by using ^original.

```
u: (_*1)
    $_tmp = ^original(_0)
    if ($_tmp != $_prior) { $_tmp}
    $_prior = $_tmp
    ^retry(RULE)
```

The above code walks each word of the sentence as seen by script. It grabs what the user actually typed (prior to any spelling corrections of any kind) and if it is different from the prior word, it prints it out. Multiple words in a sentence may all result from the same correction, so testing against the prior correction derivation is needed.

And you can also grab %originalInput for the entire raw text of what the user typed for this volley (no oob included).

Adding and subtracting meaning

Sometimes it's not that the word is wrong, it's just that words have multiple meanings and in some context you don't want to detect such a meaning. For that I often use ^mark and ^unmark. For example, if I am going to detect a country, I don't want "guinea pig" to be conflated with Guinea the country. So I can detect and kill off the ~country mark on the word.

```
u: NOCOUNTRY (_guinea pig) ^unmark(~country _0) ^retry(RULE)
...
u: COUNTRY (_~country) ...
```

Sometimes I will remove a concept, or sometimes I will hide the word itself.

```
u: NOPHONE(_phone [number \# ]) ^unmark(* _0)
```

I might have code looking for products, of which a phone is one. But this reference is not the product, and I just kill the word entirely. A single rule suffices to fix detecting this. But if one were using machine learning to locate things, you'd need lots of training sentences to get it to avoid this mistake.

Variant keywords again

Despite all the clever fancy stuff, sometimes you just have to use variant keywords. This is specifically true when the misspelling is a word itself.

```
u: ([break brake] pad) You want brakes for sure.
```

Dynamic spellchecking: ^spellcheck(input dictionary)

Not all input can be matched against prewritten scripts. It is quite possible that a bot will download a dynamic menu from a web api and try to match the user's input against that. And the menu entries may not be in the CS dictionary. For that, you need dynamic spellchecking. `^spellcheck` is given a user input and a JSON array of words or phrases to match against. It will use the basic built-in spellcheck of CS with the given dictionary to return a revised input, that could then be used with ^analyze to continue matching against the menu choices.

Interjections

A complete sentence has a subject and a verb (for commands the subject is an implied "you"). But there is another kind of sentence that lack even a verb. It is the interjection. An interjection generally starts at the beginning of the sentence. It either expresses emotion - "Oh? Are you going" where surprise is the emotion. Or it is a dialog act. "yes" (express agreement), "no"

(express disagreement), “hello” (initiate conversation), and “goodbye” (terminate conversation) are examples of dialog acts.

Old-style Interjections

In ChatScript, interjections began as spelling substitutions. The word or phrase (like “see ya later”) was replaced with the fake concept set label (~emogoodbye). If interjection splitting is turned on (by default) then the interjection became about standalone sentence and anything after it became a new sentence. Thus these two sentences look the same to CS scripts.

```
no, I can't go.     -- changes to look like below
no. I can't go.
```

This made things uniform and allows you to handle and discard interjections easily. But it was confusing because people wouldn’t know when input words would be replaced. “Beyond a doubt I like apples” becomes “~yes” and “I like apples”.

New-style Interjections

Nowadays you can choose to disable interjection processing and yet still see them where you want. You disable the interjection substitution via your bot macro:

```
$cs_token = #JSON_DIRECT_FROM_OOB | #DO_INTERJECTION_SPLITTING | #DO_SUBSTITUTE_SYSTEM | #DO
$cs_token -= #DO_INTERJECTIONS
$cs_token -= #DO_TEXTING
```

and thereafter all entries in the interjections.txt file of LIVEDATA/ENGLISH/Substitutions do not change your inputs. Texting has a mix of things including a bunch of interjections, hence turning that off.

Instead, they now also build up a legitimate concept set (like ~yes) that you can match just as you did before. The difference is the the original input words are still there. These concept sets are special, because the match is valid only at the start of a sentence and may also require they end at end of sentence or at a comma.

So “Beyond a doubt I like apples” will remain a single sentence, and the words will remain unchanged.

```
u: (_~yes _*)  This  matches.
```

_0 will bind to “beyond a doubt”. _1 is “I like apples”.

Note: Since the sentence is not changed to start with a concept name (interjection name) interjection splitting has nothing to do. It also means that multiple interjections in a row will not actually be the same. “Damn, no, I won’t go.” in old-style becomes “~emocurse. ~no. I won’t go.” But in new-style it remains unchanged. You can still match an interjection just by doing

```
u: (~emocurse) ...
```

But if you want to treat the interjections as separate sentences you can do:

```
u: ( _~no _*)    do something, then
    if (_0) {^input('_0)}
    ^fail(SENTENCE)
u: ( _~emocurse _*)  do something, then
    if (_0) {^input('_0)}
    ^fail(SENTENCE)
```

This new-style interjection is much more flexible in how you process it. You can do:

```
u: (_~no _*)  do something for the no, Then
    ^unmark(_0 *)
```

After handling the no, the above simply hides the interjection and the system now only sees whatever else was input.

Rule-based spellchecking

Sometimes users join words together in ways spellcheck can't readily detect. But you can write scripts to do it.

Replacing 1 word with multiple words

In the context of products with brands and models, sometimes user join them with hyphens. If the pieces are already in the dictionary, CS will split them automatically. But if not, you can do so manually. The trick is to be able to add extra words to the sentence.

```
u: (_*1 _10:=_0) # e.g. Epson-566
    if (^pos(preexists '_0)){^retry(TOPRULE)}
    if ( _0 != unknown-word AND ^pos(preexists _0) ){^retry(TOPRULE)}
    _5 = ^burst('_0 -)
    if (!_6) {^retry(TOPRULE)}
    $_brand = ^pos(uppercase _5)
    $_model = _6
    if (!^pos(ismodelnumber $_model) AND !^pos(isinteger $_model) ) {^retry(TOPRULE)}
    if ($_brand !? ~brands){^retry(RULE)}

    if (PATTERN @_10- _* ) {$_start = '_0}
    if (PATTERN @_10+ _* ) {$_end = '_0}
    $_sentence = ^join($_start  " "  $_brand  " " $_model " " $_end )
    $_sentence = ^substitute(character $_sentence _ " ")
    ^analyze($_sentence)
    ^retry(TOPRULE)
```

This code splits a word at a hyphen if it can (assuming the word is not already in the dictionary). It then confirms the model is a potential model number and the brand is a known brand. If so, then it gets the sentence fragments before and after this word and forces an in-place reanalysis of this new sentence (which now has the proper words) and then continues merrily hunting for more and then continuing with the rest of analysis.

The above was easy because you could split on a hyphen. Otherwise you can maybe do this:

```
u: (_*1 _10:=_0) # check for Epson566
    if (^pos(preexists '_0)){^retry(TOPRULE)}
    if ( _0 != unknown-word AND ^pos(preexists _0) ){^retry(TOPRULE)}
    @0 = ^burst('_0 "") # character burst
    Loop()
    {
        $_char = ^first(@0subject)
        if ($_char ? ~digits OR $_remainder)
        {
            $_remainder = ^join($_remainder $_char)
        }
        else {$_word = ^join($_word $_char)}
    }
    $_brand = ^pos(uppercase $_word)
    $_model = $_remainder
    if (!^pos(ismodelnumber $_model) AND !^pos(isinteger $_model) ) {^retry(TOPRULE)}
    if ($_brand !? ~all_tech_brands){^retry(RULE)}

    if (PATTERN @_10- _* ) {$_start = '_0}
    if (PATTERN @_10+ _* ) {$_end = '_0}
    $_sentence = ^join($_start  " "  $_brand  " " $_model " " $_end )
    $_sentence = ^substitute(character $_sentence _ " ")
    ^analyze($_sentence)
    ^retry(TOPRULE)
```

Alternate meanings without rules

For some texting, we replace words with their equivalents, like ‘cu’ becomes ‘see you’. But this spell check change directly impacts words you see and match in the input. It’s not necessarily always appropriate. For example ‘b’ means ‘be’ in texting, yet we may not want to lose the meaing for the letter, as in “Vitamin B”. In such cases you can make the letter mark the word using a rule, as mentioned earlier, like:

```
u: (_b) ^mark(be _0) ^retry(RULE)
```

This works, but has some small execution time cost. You can make this happen

automatically without the rule, if you arrange to make a member fact out of it. That is, we can pretend "be" is a concept.

```
table: ^texting($_letter $_meaning)
    ^createfact($_letter member $_meaning)
data:
b be
```

This means inputs with the letter b, will see and mark meanings of the letter, including marking 'be' and everything that 'be' is a member of. It will not, however, mark parts of speech of 'be' or have it show up in parse data since parsing and pos-tagging was done on the letter, not the word.