ChatScript Common Beginner Mistakes

© Bruce Wilcox, gowilcox@gmail.com brilligunderstanding.com

Revision 8/12/2017 cs7.53

Here are a collection of common beginner mistakes I have seen. Many will work, they are just unnecessary and inefficient and potentially more obscure to read.

Failing to separate tokens with a space

```
u:MYLABEL()
```

is not the same as `u: MYLABEL()` While ChatScript will automatically space around balanced things like parens or brackets, it will not do so around ordinary text, so `u:MYLABEL` is a single token, an ordinary word for printout, and not a rule header.

Leading and trailing * in a pattern

```
u: (* I go home *)
```

The above pattern you might see in AIML, which requires a pattern that covers the entire sentence. CS does not and the *'s are superfluous. CS will hunt into the sentence to find the first pattern element. And it will finish matching when it has matched the last pattern element. CS does not require you cover the rest of the sentence. Therefore this is sufficient (and faster to execute and takes less space and is easier to read).

```
u: (I go home)
```

Inadequate understanding of rejoinders

```
t: How old are you?
    a: (~number<5) You are young.
    b: (~number<10) You are a growing child.
    c: (~number<100) You are old.
```

The rejoinders above are all different responses to the same input. They should be at the same level, meaning all should be `a:`, which name the alternatives at the same level. And, to be clear, I would intent them.

```
t: How old are you?
    a: (~number<5) You are young.
        b: (~no) You aren't young?
    a: (~number<10) You are a growing child.
```

```
    a: (~number<100) You are old.
```

The above makes it clear that the `a:` level rejoinders are tied to the `t:`, and the `b:` rejoinder is tied to the `a: (~number<5)`. If you don't indent clearly, then

```
t: How old are you?

u: (~number<5) You are young.
```

it is hard to tell if the `u:` is intentional or a typo. I left-flush all top level rules (`t:`, `u:v,?:,s:‘`) and appropriately indent based on nesting level all rejoinders. Structurally it is immediately obvious what is intended.

Failing to use sample input comments

Whenever you have a responder or rejoinder, you should have one or more sample inputs above it.

```
##<<Supports the following formats
december 2, 1980
december 2 in 1980
dec second of 1980
##>>
u: ( * _~month_names {, } _~number { in of} _{~yearnumber} )
```

The above rule was elegantly documented with a comment. But it is not appropriate. The correct way is

```
#! december 2, 1980
#! december 2 in 1980
#! dec second of 1980
u: ( * _~month_names {, } _~number { in of} _{~yearnumber} )
```

The difference is- the comment expressed the scriptors intent of what they were doing. But it will take work for anyone looking at the code (including the author) to decide that the pattern actually works for those cases. And you will probably get it wrong. When you write sample inputs, CS can be asked to :verify that the pattern does indeed work for the cases given and tell you. Not only now, but anytime in the future, when other parts of the system evolve and may impact your pattern accidently (changes in concept sets for example). Writing sample inputs is both documentation so that people can understand your script without having to interpret the pattern, and it is the unit test of a rule, allowing the system to verify all sorts of things (not just that the pattern works).

Omitting ^ from a function call

```
u: (_~number) $$tmp = nth(~set _0)
```

The above code is legal. The system will figure out that nth is an existing function `^nth` and call it appropriately. Yet you should avoid being sloppy and always put the ^ in front of a function call. Why?

If `nth` is not a function, because CS intermixes english and script, CS does not know you intended a function call and will merrily treat it as ordinary text with no warning or error. If, however, you write `^nth`, CS will confirm that it is a function and issue an error if it is not. This error protects you from typos, mistyping the name of a function.

Transferring function values to variables

```
outputmacro: ^func(^arg1 ^arg2)
$$units = ^arg1
$$time_value = ^arg2
if ($$units == weeks)
```

You can name function arguments whatever you want. And you can use them directly in your code (you just can't overwrite them via assignment). So it is clearer and cleaner to do this:

```
outputmacro: ^func(^units ^time_value)
if (^time_value == weeks)
```

Fail to take advantage of canonical values of keywords

```
u: ( _~number [second seconds ] )
```

The above [] contains redundant information. Second is the singular canonical form of seconds. You don't need both words. Which means you could instead just write

```
u: (_~number second)
```

Excessive topic keywords

Typically no topic should have pronouns, prepositions, conjunctions, and determiners as keywords.The goal of keywords in a topic is to help CS find relevant topics. When you look at a topic, the keywords should all be immediately suggestive of that topic.

```
topic: ~childhood (my childhood young)
```

The keyword my doesn't immediately bring this topic to mind and should be removed. It will force CS to consider this topic for all sorts of sentences that contain my that have nothing to do with childhood. childhood is clearly a good keyword. Is young? Probably. Maybe.

Redundant {} choices

The purpose of the optional words list is to assist in aligning words in a pattern. If it doesn't matter whether or not the `{}` exists in the pattern, then the optional choice is wasted.

```
u: ( I love you {maybe} )
```

In the above pattern, there is nothing after `{maybe}`. This means that the pattern won't care if a match happens or not for that.

```
u: ( I will love you {maybe} tomorrow)
```

Here, the optional `{maybe}` is valuable, allowing the pattern to match both *I will love you tomorrow* and *I will love you maybe tomorrow*. That is, it swallows up a useless word to allow the real match to proceed. The above was overly specific, and would more likely be

```
u: ( I will love you {~adverb} tomorrow)
```

Optional matches can be superfluous or downright detrimental at the start.

```
u: ( {~adverb} I will love you)
```

The intent of the above was to swallow up an adverb occurring before I, like Maybe I will love you. But, firstly, it wasn't needed. CS would have skipped any words until it got to I anyway. Secondly, it will prove detrimental if the input is I will soon love you, because it will scan forward, find the soon as an adverb, lock the position of matching to that, and then look for I immediately afterwards. But now it's too late and I has already gone past.

My pattern doesn't match the input I crafted it for!

Probably true. First thing to do is

```
:prepare this is my special input
```

Look at the marked concepts for each word. Does it match your pattern? Is the input even what you expect? Has CS told you it substituted something or spell-fixed it or whatever?

Wrong understanding of scoping of variables

```
outputmacro: ^myfunc(^myargument)
```

Function arguments like ^myargument only exist during the code of the macro and will not be legal elsewhere. Other than function arguments, all other variables have global scope. This means if you do

```
$$tmp = 1
```

in some topic, it will have that value afterwards in every topic, and you risk destroying it if you do an inadvertent assignment in some other topic of

```
$$tmp = hello
```

Separating keywords with commas

```
Topic: ~mytopic [keyword1, keyword2, keyword3]
```

is wrong. Don't use commas.

Listing phrases or titles in [] instead of quotes

```
u: ([Doctor Zhivago])
```

is wrong. First, [] means find one word of. Second, for titles of things, you should put them in double quotes. `u: ("Doctor Zhivago")` The double quotes tells CS that you think of this as a single item, not two words. It's not like `u: (have fun)`, where you want the words together but they are not a single item. Proper nouns should always be a single item, particularly when there is punctuation in the item like "Des Moines, Iowa" because you don't really know how CS manages punctuation internally- does it split the comma into a separate token, does it use underscores, etc. You shouldn't have to worry about how CS represents it. You should just double quote it.

Being unaware of interjections in CS

```
topic: ~introductions( hi hello welcome)
```

is wrong because hello is not a normal word like a noun, verb, adjective, or adverb. It is a special word that is a "dialog act" or an "interjection". CS auto translates interjections on input. If you do

```
:prepare hello
```

you will see it decodes it into `~emohello`, and that's what you should use as your keyword. The most common interjections are `~emohello`, `~emogoodbye`, `~emohowzit`, `~yes`, `~no`.

Expecting CS numeric assignment to mirror other languages

ChatScript executes as it goes, it does not stack up data like a desk calculator. So

```
$tmp = %hour + $tmp
```

is executed as: `$tmp = %hour` and then `$tmp += $tmp`. This is not what you expect from other languages, where you expect `(%hour + $tmp)` to be computed and then assigned. You can, of course, make your code simpler and correct by

```
$tmp += %hour
```

Or even

```
$tmp = $tmp + %hour
```

Thinking [] in a pattern will find the earliest match

A pattern like `u: ( the * [bear raccoon] ate )` contains a list of words in `[]`, that a human thinks of as equivalent, and thus imagines CS will simply find the next occurrence of any of them in the input. For efficiency, that's not how CS works. It will try each in turn, and if a match is found, it will stop trying words in the `[]` immediately. So input like: *the raccoon ate the bear* will fail because bear will be found first, and *ate* does not happen immediately after that. If you convert `[]` into a concept set, then a pattern of `u: (the * ~myanimals ate)` will match, because all members of a concept set are considered simultaneously.