ChatScript Coding Standards

Contents:

Rationale: Coding Standards allow you and others to understand your code. They can also allow others who are not programmers to view an abstract of your code, generated by CS's `:abstract` ability.

To use `:abstract`, merely login locally with a new user name (or erase contents of USERS), then type `:abstract ~mytopic` for a single topic. The abstract will whiz by on screen but is captured in the user log. Just rename that log file and move it somewhere else.

Indentation of rules

- Do not indent top level rules.
- Indent rejoinders per level.
- Leave blank line before top level rule.

Examples:

```
t: How are you
    a: (~goodness) glad to hear it.
        b: (cool) not so cool.
    a: (~badness) sorry to hear it
    b: (cool) not so bad. # bad - wrong indentation level

u: (how are you) I am good.
u: (and then) xxx # this is bad - jammed immediately after top level rule

    u: (why not) because. # this is bad indentation of top level rule
```

Rationale: On one hand you want to minimize useless space on a page. So top-level rules should not be indented (nor should their sample inputs).

On the other hand you want to easily see the structure of the code, so properly indenting rejoinders makes structure clear. And you want to easily spot where one rule chunk ends and another begins. So separating top level rules and keep rejoinders together is what I do.

Rule Labels

- Make meaningful labels using capital letters.
- Label all rules that have their own output text.

Examples:

```
t: ASK_FOR_EMAIL () What is your email address?
    a: EMAIL_GIVEN (~email_url) Thank you.
    a: () ^reuse(EMAIL_GIVEN)

t: B55() What is your email address? # bad- meaningless label
```

Rationale: Using capital letters for rule labels (both at the rule and in places that `^reuse` the rule) make it easy to see them in a mass of text like a `:trace`.

If yours is not your own private bot project then every rule that generates output should have such a label, which will appear in an abstract. This allows others to refer to your rule and maybe find it in log files of customers.

Give sample inputs

- Give sample inputs for responders and rejoinders.
- Give multiple samples for wildly different sentence constructions when you have multiple patterns in a rule.

Examples:

```
t: What year is it?
    #! 1993
    a: (~year) Great.

#! What is your name
#! Who are you?
?: ([
    (‘you [name called])
    (who be ‘you)
    ])
    My name is Rose.
```

Rationale: Sample inputs explain your patterns. Instead of having to interpret the pattern, you know immediately what the rule is intending to do.

Sample inputs allow you to use `:abstract` to give non-programmers an overview of your code. Sample inputs also allow CS to unit-test your code for you using `:verify`.

Easy-to-read patterns

- Avoid superfluous parentheses.
- For multiple patterns, put each on its own line.

Examples:

```
?: WHAT_BRILLIG_DOES([
    (what * "Brillig Understanding" )
    ([tell talk know] *~2 about "Brillig Understanding")
    ])
    Brillig Understanding makes natural language systems.

u: BAD_CHOICES ([
    (<< Brillig company >> ) # superfluous parens
    (~mywords) # superfluous parens
    ([ word1 word2]) # superfluous parens
    ])
```

Rationale: Superfluous parens make patterns harder to read, so harder to visually confirm they are correct. They also slow down the CS engine, but not enough to matter.

When you are using multiple patterns in a rule, putting each on its own line allows you (or code reviewers) to comprehend each one separately.

Easy to read rule output

Indent each sentence and each CS code statement on separate lines.

Examples:

```
#! Test
u: (test)
    $status += 1
    $_x = ^compute(1 + 2)
    The answer is $_x.
    Did you miss it?

#! Where were you born
u: (where * you * born) # bad- harder to read
    I was born in San Francisco near the old church on the hill. I was born an
    only child but my parents always wanted more.
```

```
#! Where were you born
u: (where * you * born) # good
    I was born in San Francisco near the old church on the hill.
    I was born an only child but my parents always wanted more.
```

Rationale: Multiple sentence user output is harder to read if on the same line since they will tend to run into very long lines.

Code is certainly harder to understand when multiple actions are on the same line. On the other hand if the output is tiny, you might put it on a single line like this:

```
u: (test) OK. $status += 1
```

Bundle related rules

- Put related rules in topics
- Put more closely related rules together when you can.

Examples:

```
topic: ~aliens()
...

topic: ~family()

#!x*** Parents

#! Who is your mother
?: ( who * mother) Mom.

#! What does your mother do?
?: ( what * mother * do) She works.

#! Who is your father # note we had all mother stuff together before father
?: ( who * father) Dad.

#! What does your father do?
?: ( what * father * do) He works.
#!x*** Siblings
...
```

Rationale: With appropriate bundling you should immediately know where to add a new rule or discover that you have multiple rules doing the same job.

I even annotate clusters of rules for `:abstract` using `#!x` comments which will reflect into the abstract to label the cluster and make it easy for people to follow

along. The only problem may be when some earlier rule blocks a later one and you need to move something around. Consider using `!` keywords as instead.

Concept and Function localization

- If a concept or function is only used in 1 file, put it into that file.
- If a concept or function is used in multiple places, use a global functions.top or concepts.top.
- For tables used only used in 1 file, depends on size of table.

Rationale: When you can localize data to the one file that uses it, it makes it easier to find the definition when you want to inspect it.

Keyword Casing and Misspellings

Use standard dictionary casing (or product owner 's casing in the case of new non-words in the language) for keywords in concepts or patterns.

Avoid putting misspellings in concepts or keywords. Use replace: in your scripts file to effectively add additions to the predefined substitutions files that come with ChatScript.

But use `:replace` only when the misspell has only one possible obvious value. `replace:` changes the dictionary for ALL BOTS.! For misspellings which are legal words, add them to concepts or keywords.

Examples:

```
concept: ~protocols (Blu-ray WiFi)
replace: blue_ray Blu-ray # true always. Do this as the correct spelling
replace: MS Microsoft     # bad - computers true, medical not

#! I used my brakes
u: ([brake break]) Don't hit the brakes too hard. # misspell is a real word, list it

#! I like mars
u: (_~planets) '_0 # outputs Mars even though user typed mars

#! Do you like sheep?
u: ([sheep shep sheap]) I like sheep # generally bad to do this
```

Rationale: If a word exists with only one casing in the dictionary, CS can automatically adjust the input to make it correct. Even if a word exists in multiple cases, if a concept set has it in only one casing (e.g. `~planets` has the noun Mars and not the verb mars), if you memorize it, CS can memorize the correct casing.

Sometimes you expect some user typos. Putting the illegal forms in your patterns (like shep), means they enter the dictionary as acceptable words. This means spelling correction can change "misspellings" of misspellings, causing word drift for an input like "hep". Prefer to use `replace:` if a misspelling can only go to one obvious place.