ChatScript Debugging Manual

Revision 10/16/2022 cs12.3

You've written script. It doesn't work. Now what? Now you need to debug it, fix it, and recompile it. Debugging is mostly a matter of tracing what the system does and finding out where it doesn't do what you expected.

Sometimes debugging involves changes to the cs_init.txt and cs_initmore.txt files. In stand-alone mode you can just type `:restart` after editing those files and cs will reboot. In a live running cs however, if you have not authorized debug commands then you can't type restart. But you can add a top level file named `restart.txt` which will force cs to reboot itself as it starts to process the next user input (and it erases the file after doing that).

If you don't have Windows, then debugging mostly done by issuing debug commands to the engine, as opposed to chatting. If you have windows, you can run your program under ChatScriptIDE to debug it interactively. See the ChatScript-Debugger manual.

Debugging commands only show their data if user logging is enabled. In cs_init.txt you can put 'userlogging-file' or create at top level the file 'userlogging.txt'.

If the system detects bugs during execution, they go into `TMP/bugs.txt` You can erase the entire contents of the TMP directory any time you want to. But odds are this is not your problem.

Debugging generally requires use of some :xxxx commands. I don't always remember them all, so at times I might simply say

Special Input controls in user input

```
#!Rosette#my message
```

The use of the above syntax means to change the bot to Rosette and then use `my message` as the actual input.

:commands

to get a list of the commands and a rough description.

```
:commands
```

the above : statement show the list:

```
---- Debugging commands -
:do        - Execute the arguments as an output stream, e.g., invoke a function, set variabl
:silent    - toggle silent - dont show outputs
:log       - dump message into log file
:noreact   - Disable replying to input
:notime    - Toggle notiming during this topic or function
:notrace   - Toggle notracing during this topic or function
:redo      - Back up to turn n and try replacement user input
:retry     - Back up and try replacement user input or just redo last sentence
:say       - Make chatbot say this line
:skip      - Erase next n gambits
:show      - All, Input, Mark, Number, Pos, Stats, Topic, Topics, Why, Reject, Newlines
:time      - Set timing variable
             (all none prepare match ruleflow pattern query json macro user usercache sql to
:trace     - Set trace variable
             (all none basic prepare match output pattern infer query substitute hierarchy f
:why       - Show rules causing most recent output
:authorize - Flip authorization for all debug commands
:comparelog - given path/names of 2 log files, show entries that are different
:ingestlog - given name of log file, reexecutes it and reports differences from log

---- Fact info -
:allfacts  - Write all facts to TMP/facts.tmp for current bot current language.  :allfacts a
:facts     - Display all facts with given word or meaning or fact set
:userfacts - Display current user facts

---- Dict info -
:alldict - Lists all words in dictionary, and if called with :alldict fact, lists the facts

---- Topic info -
:gambits    - Show gambit sequence of topic
:pending    - Show current pending topics list
:topicstats - Show stats on all topics or named topic or NORMAL for non-system topics
:topicinfo  - Show information on a topic
:topics     - Show topics that given input resonates with
:where      - What file is the given topic in

---- System info -
:commands        - Show all :commands
:context         - Display current context labels
:conceptlist     - Show all concepts- or with argument show concepts starting with argument
:definition      - Show code of macro named
:directories     - Show current directories
:functions       - List all defined system ^functions
:identify        - Give version data on this CS
```

```
:macros          - List all user-defined ^macros and plans
:memstats        - Show memory allocations
:list            - $ (variables) @ (factsets)  _  (match variables)  ^ (macros)  ~ (topics&c
:queries         - List all defined queries
:timedfunctions  - List all user defined macros currently being timed
:timedtopics     - List all topics currently being timed
:tracedfunctions - List all user defined macros currently being traced
:tracedtopics    - List all topics currently being traced
:variables       - Display current user/sysytem/match/all variables
:who             - show current login/computer pair

---- Word information -
:down      - Show wordnet items inheriting from here or concept members
:concepts  - Show concepts triggered by this word
:findwords - show words matching pattern of letters and *
:hasflag   - List words of given set having or !having some system flag
:nonset    - List words of POS type not encompassed by given set
:overlap   - Direct members of set x that are also in set y somehow
:up        - Display concept structure above a word
:word      - Display information about given word
:dualupper - Display words that have multiple upper-case forms

---- System Control commands -
:build   - Compile a script - filename {nospell,outputspell,reset}
:bot     - Change to this bot
:crash   - Simulate a server crash
:debug   - Initiate debugger
:flush   - Flush server cached user data to files
:language - set current language to one names on the command line param language=. or returr
:quit    - Exit ChatScript
:reset   - Start user all over again, flushing his history
:restart - Restart Chatscript
:user    - Change to named user, not new conversation

---- Script Testing -
:autoreply   - [OK,Why] use one of those as all input.
:common      - What concepts have the two words in common.
:prepare     - Show results of tokenization, tagging, and marking on a sentence
:regress     - create or test a regression file
:source      - Switch input to named file
:tsvsource  - switch conversation to a named file
:concat - switch to file where multiple lines are joined to a single input line
:testpattern - See if a pattern works with an input.
:testtopic   - Try named topic responders on input
:verify      - Given test type & topic, test that rules are accessible.
               Tests: pattern (default), blocking(default), keyword(default), sample, gambit
```

```
:verifylist, :verifyrun, :verifymatch - Tests system to show rules are accessed when outside

---- Document Processing -
:document - Switch input to named file/directory as a document {single, echo}
:wikitext - read wiki xml and write plaintext
:tsv    - convert a tab-delimited spreadsheet into CS table format, with double quotes arou

---- Analytics (see analytics manual) -
:abstract     - Display overview of ChatScript topics
:coverage     - Save execution coverage of ChatScript rules
:showcoverage - Display execution coverage of ChatScript rules
:diff         - match 2 files and report lines that differ
:trim         - Strip excess off chatlog file to make simple file TMP/tmp.txt
:timelog      - read a log file (like system log) and compute average, min and max times of
:splitlog      - read a log file (like system log) and create csv of input, output, bot, why

---- internal support -
:spellit          - given sentence provide explanation for spelling corrections
:topicdump         - Dump topic data suitable for inclusion as extra topics into TMP/tmp.txt
                     (:extratopic or PerformChatGivenTopic)
:builddict         -  basic, layer0, layer1, or wordnet are options instead of default full
:buildforeign      - rebuild a foreign dictionary (requires treetagger data)
:clean             - Convert source files to NL instead of CR/LF for unix
:extratopic        - given topic name and file as output from :topicdump,
                     build in core topic and use it thereafter
:pennformat        - rewrite penn tagfile (eg as output from stanford) as one liners
:pennmatch         - FILE {raw ambig} compare penn file against internal result
:pennnoun          - locate mass nouns in pennbank
:pos               - Show results of tokenization and tagging
:sortconcept       - Prepare concept file alphabetically (see writeup)
:translateconcept  - TODO
:timepos           - compute wps average to prepare inputs
:verifypos         - Regress pos-tagging using default REGRESS/postest.txt file or named fil
:verifyspell       - Regress spell checker against file
:verifysubstitutes - Regress test substitutes of all kinds
:worddump          - show words via hardcoded test
:verifySentence    - verification data
:labelremap     - for a topic builds an xref of internal tags to their labels (if they have
```

All commands can be written in full, or most can be abbreviated using `:firstletterlastletter`, eg

```
:build
```

can be

```
:bd
```

`:commands off` will locally disable commands until a corresponding `:commands on` is given.

Before it goes wrong during execution

Before you chat with your chatbot and discover things don't work, there are a couple of things to do first, to warn you of problems.

Compiling (`:build`)

Of course, you started with `:build` to compile your script. It wouldn't have passed a script that was completely wrong, but it might have issued warnings.

That's also not likely to be your problem, but let's look at what it might have told you as a warning. The system will warn you as it compiles and it will summarize its findings at the end of the compile. The compilation messages occur on screen and in the log file.

The most significant warnings are a reference to an undefined set or an undefined `^reuse` label. E.g.,

```
*** Warning- missing set definition ~car_names
*** Warning- Missing cross-topic label ~allergy.JOHN for reuse
```

Those you should fix because clearly they are wrong, although the script will execute fine everywhere but in those places.

```
*** Warning- flowglow is unknown as a word in pattern
```

Warnings about words in patterns that it doesn't recognize or it recognizes in lower case but you used upper case may or may not matter. Neither of these is wrong, unless you didn't intend it. Words it doesn't recognize arise either because you made a typo (requiring you fix it) or simply because the word isn't in the dictionary.

Words in upper case are again words it knows as lower case, but you used it as upper case. Maybe right or wrong.

echorule

When you add this after the build name, the system will show what rule pattern is has most recently compiled in the log. If build crashes in a topic or you are not clear where you are when things go wrong, this trace can help identify the rule in question.

ignorespell: ...

You can disable case spelling warnings for specific words by naming them in lower case like this:

```
ignorespell: me or
```

This will not warn you when the system detects `me` or `ME` (abbreviation of Maine). You can also globally turn off spell warnings using

```
ignorespell: *
```

which remains in effect until you do

```
ignorespell: !*
```

You can, for example, fix all warnings you can in a file, And once you believe there are no more issues and the file is frozen, just turn ignorespell on at start of file and off at end of file.

Editing the main dictionary is not a task for the faint-hearted. But ChatScript maintains secondary dictionaries in the `TOPIC` folder and those are easy to adjust. To alter them, you can define concepts that add local dictionary entries. The names of important word type bits are in `src/dictionarySystem.h` but the basics are `NOUN`, `VERB`, `ADJECTIVE`, and `ADVERB`.

```
concept: ~morenoun NOUN (fludge flwoa boara)
```

A concept like the above, declared before use of those words, will define them into the local dictionary as nouns and suppress the warning message. You can be more specific as well with multiple flags like this:

```
concept: ~myverbs VERB VERB_INFINITIVE (spluat babata)
```

You can define concepts at level 0 and/or level 1 of the build, so you can put define new words whenever you need to.

When build is complete, it will pick up where it left off with the user (his data files unchanged). If you want to automatically clear the user and start afresh, use

```
:build xxx reset
```

Tracing during compilation

You can put :trace commands interspersed with table arguments or between top level commands in script files. You can put :debug commands interspersed with table arguments.

After it goes wrong during execution

The most common issue is that ChatScript will munge with your input in various ways so you don't submit what you think you are submitting. The substitutions files will change words or phrases.

Spell correction will change words. Proper name and number merging will adjust words. And individual words of yours will be merged into single words if WordNet lists them as a multiple-word (like `TV_star`). So you really need to see what your actual input ended up being before you can tell if your pattern was correct or not. Thus the most common debug command is `:prepare`.

Most Useful Debug Commands

`:prepare this is my sample input`

This shows you how the system will tokenize a sentence and what concepts it matches. It's what the system does to prepare to match topics against your input. From that you can deduce what patterns would match and what topics might be invoked.

If you give no arguments to prepare, it just turns on a prepare trace for all future inputs which disables actually responding. Not usually what you want.

There is an optional first argument to `:prepare`, which is a variable. If given it indicates the `$cs_token` value to use for preparation. E.g

```
:prepare $mytoken this is a sentence.
```

The `$cs_prepass` variable contains optional topic script that will be executed before the main control script is executed. The topic should be flagged SYSTEM so it does not erase itself as it gets used.

The advantage of `$cs_prepass` is that it is also executed by `:prepare`, so you can see traces of it there. If you want to see preparation w/o this topic (raw preparation by the engine) you can add the optional argument `NOPREPASS` or `PREPASS`. Eg.

```
:prepare NOPREPASS This is a sentence
:prepare $newtoken NOPREPASS This is a sentence.
```

The use of the `PREPASS` or `NOPREPASS` is remembered during the run of CS, so that the next time you call `:prepare` you can omit it and the same setting will be used. If all you want to know is what concepts a word is involved in, consider `:common`.

`:why`

When I test my bots (assuming they pass verification), I chat with them until I get an answer I don't like. I then ask it why it generated the answer it did.

This specifies the rules that generated the actual output and what topics they came from. I can often see, looking at the rule, why I wouldn't want it to match and go fix that rule. That doesn't address why some rule I want to match failed, so for that I'll need typically need tracing.

`:clearlog`

Erases current user's log file.

`:comparelog`

`:comparelog LOGS/serverlog1024.txt  oldlog.txt`

Will read both logs in parallel and list into user log significant differences (assumption is that both logs represent the same inputs against different versions of CS).

`:ingestlog  filename`

`:ingestlog LOGS/serverlog1024.txt`

Will read named cs log file and re-execute it. Differences will be reported in tmp/ingesterr.txt as well as user log when enabled and briefly onscreen.

Trace

`:trace all / :trace always / :trace none`

The ultimate debugging command dumps a trace of everything that happens during execution to the screen and into the log file. After entering this, you type in your chat and watch what happens (which also gets dumped into the current log file). Problem is, it's potentially a large trace. You really want to be more focused in your endeavor.

But ignoring that,

`:trace all`

turns on all tracing. It can be suppressed in areas of code executing within `^NOTRACE()`.

`:trace always`

ignores ^notrace protection and traces everything.

`:trace ignorenotrace`

allows you to use the limited traces below, and still ignore `NOTRACE` covered calls.

If you want to jump to where the user output was created and then see where you are, search for "message:".

There are fake system variables you can execute in output code to turn on and off debugging also. These are %trace_on and %trace_off. They are simpler to remember how to do than invoking ^eval to invoke debug commands. And these disallow behavior if server authorization doesnt exist. These variables do not actually show up in output.

```
:trace ^myfunction
```

this enables tracing for the function. Call it again to restore to normal.

```
:trace $var
```

this enables tracing when this variable changes. Local variables only show a trace during the current volley. Global variables show a trace across volleys.

```
:trace ~education
```

this enables tracing for this topic and all the topics it calls. Call it again to restore to normal. This is probably what you need usually to see what went wrong. You know the code you expected to execute, so monitor the topic it is in.

```
:trace !~education
```

this disables current tracing for this topic and all the topics it calls when `:trace all` is running. Call it again to restore to normal.

Similarly you can do

```
:trace ~education.4.5
```

which says to trace just the 4th top level rule in `~education`, rejoinder # 5. The above defaults to tracing everything while in that topic. You can specify what you want to trace by naming trace values and then the topic.

```
:trace prepare basic facts ~education
```

The above will turn on a bunch of trace values and then assign them to the topic, clearing the trace values.

```
:trace input
```

:trace prepare can generate a lot of data about concept marking. If you just want a simple check on what the adjusted input looks like, you can do `:trace input`

```
:trace prepare basic facts ^myfunc
```

The above will turn on a bunch of trace values and then assign them to the function, clearing the trace values.

Note: If you want to set a regular trace value and a topic and/or function, the regular trace values must occur last, after all topic and function traces.

```
:trace factcreate subject verb object
```

subject, verb, and object name values of those fields. You can use null when you don't want to name a value. This trace will show whenever facts matching this template are created. You can insert a trace command in the data of a table declaration, to trace a table being built (the log file will be from the build in progress). E.g.,

```
table: ~capital (^base ^city)
_9 = join(^city , _ ^base)
^createfact(_9 member ~capital)
DATA:

:trace all
Utah Salt_lake_city
:trace none
```

You can insert a trace in the list of files to process for a build. From files1.txt

```
RAWDATA/simplecontrol.top # conversational control
:trace all
RAWDATA/simpletopic.top # things we can chat about
:trace none
```

And you can insert a trace in the list of commands of a topic file:

```
topic: foo [...]
...
:trace all
```

Tracing is also good in conjunction with some other commands that give you a restricted view. You can name tracing options by subtraction. E.g.,

```
:trace all -infer -pattern
```

When I'm doing a thorough trace, I usually do

```
:trace all -query
```

because I want to see fact searches but only need the answers and not all the processing the query did. If you want to see the arguments and answer to a query, you can leave :tracel all, but then do :notrace ^query. This will shut down all the internal guts of the ^query function.

NOTE: if you want tracing to start at startup, when you don't have control, login with a botname of trace. E.g, at the login type:

```
john:trace
```

This will turn on all tracing.

`:trace` also has individual collections and items it can trace. If you just say :trace you get a listing of things.

```
:notrace {ON,OFF} ~topic1 ~topic2 {ON,OFF} ~topic3
```

Regardless of the use of `:trace`, `:notrace` marks some topics to not trace. Particularly useful to do :notrace ^Query which omits most of the guts of query calls, which can often be extensive.

By default, the flag is set `ON` on listed topics but you can change the value on the fly to enable trace blocking on some topics or turn it off on ones previously turned on.

You can also trace specific kinds of data. If you just say `:trace`, it will list the status of what is and is not being traced. Calls to `:trace` can be additive, you can name some and add more in another call. Here are things you can trace:

how to trace	description
match	traces the rule finding/matching (overlaps with pattern but is more abstract)
variables	at start of every volley display permanent variables read in
simple	match + variables

how to trace	description
ruleflow	show labels, some of the pattern and some of the output code of a rule whose
output	is executed. It's a very abbreviated view of where flow of control went
input	shows new input submitted via ^input
prepare	trace sentence preparation
output	traces input AND output coming in and going out normally
pattern	trace pattern matching processor
mild	prepare + output + pattern
infer	traces functions that manipulate facts apart from query and fact

how to trace	description
sample	when testing rules, show the sample input as part of trace (easier to see where you are)
substitute	trace substitution system
hierarchy	internal showing of marking process (even more detailed if its the only flag)
fact	traces creation and killing of facts
varassign	show variables changing state
query	trace query calls (turning off removes a lot of noise)
user	trace load/save of user topic data
pos	trace pos tagging
tcp	traces tcpopen calls
json	traces jsonopen calls

how to trace	description
macro	trace user-defined macro calls with arguments and return values
usercache	internal showing of user topic file caching
sql	traces postgres calls
label	trace match results on rules with labels
treetagger	show tt behavior
topic	erasing rules, issues in pushing a topic, rejoinder setting
deep	input macro sample infer substitute fact varassign query user pos tcp usercache sql label topic

Example:

```
:trace all -query -infer
```

If you want to trace a topic in a limited manner, set the limits before naming the topic:

```
:trace pattern pos ~mytopic
```

The system will set the limits, then upon hitting the topic name, swallow them to be used for that topic. This means you can do multiple topics and global limits in one command:

`:trace pattern ~mytopic pos ~yourtopic varassign`

`~mytopic` gets the pattern limit, `~yourtopic` gets the pos limit, and globally you do varassign.

Note: the correct way to turn on/off normal tracing from script is by changing the tracing variable, e.g., `$cs_trace = -1`.

Tracing a server

Sometimes you do not control the user account, so you cannot do :trace from it. You can do `:trace universal` from any account to force the server to trace all users. Turn it off with `:trace none`. Obviously not a good thing to do on a highly active server as it slows everyone and generates large logs potentially.

Also you can do :serverlog 1 to turn on server logging and :serverlog 0 to turn off serverlogging. You can even force server logging without restarting a server by merely adding the file "serverlogging.txt" with arbitrary contents to the top level of CS and removing when you are done. This is actually what :serverlog does.

Similarly, if file `traceboot.txt` exists at the top level of CS when the server executes any boot code, that will be traced (same as adding `traceboot` in command line params).

Creating the top level file `prelogging.txt` turns on prelogging into server and user files (when logging into those files).

Understanding a pattern trace

```
try 3.0: ( ! %response=0 !%lastquestion !%outputrejoinder )
        ( !%response(0)=0- Remaining pattern: !%lastquestion ! %outputrejoinder )
      try 6.0: ( < do you know _a _* )
        ( <+ do- Remaining pattern: you know _a _* )
      try 7.0 DESCRIBE: ( !describe < { tell_me explain } what is it like _* )
        ( !describe+ <+
          { tell_me- explain- }+
          what- Remaining pattern: is it like _* )
      try 8.0: ( < what is _*~4 like > )
        ( <+ what- Remaining pattern: is _*~4 like > )
```

When the system is trying responders and rejoinders, for each rule it says "try" and names which top-level rule it is, and perhaps what rejoinder.

3.0 means the 4th top level rule.

The .0 refers to the rejoinder underneath it, but 0 means the rule itself. Along with the rule it, it may put out the label if there is one (eg 7.0 DESCRIBE) and it puts out the pattern it will try to match.

The next line begins matching data. Each pattern element will be marked at the end of it with a `+` if it matched, or a `-` if it didn't match.

Whenever the system enters a paired token, i.e., `()`, `[]`, `{}`, `<< >>`, it indents and moves to a new line. When it finishes a paired token, it will mark with a + or - whether the pair worked and it will resume the old indentation as it continues.

When a relationship test is performed, the value of a piece of it will be given in parens. Similarly when a concept is matched, the matching word will be shown in parens.

When a pattern fails, the system will say "remaining pattern" and show what it has not tried to process. Occasionally when an initial word matches but the rest do not, the system will say —— Try pattern matching again, after word 1 (where)

`:tracedfunctions`

list all user functions being traced

`:tracedtopics`

list all topics being traced

`:time`

measures milliseconds taken by topics and functions for performance analysis. Just type :time to see keywords.

`:time all` turns on timing everywhere

It is similar to `:trace` (using several of the same keywords) and can be used to limit to just certain topics/rules and ^macros. A simple `:time` will list those keywords.

Note the time output is only shown in the user's log file and is not echoed to the console.

By default items that don't take any time will not be shown, but this can be overridden by `:time always`

:notime

disables timing for some topic or functions

:timedtopics

lists what topics are currently being timed.

:timedfunctions

list functions currently being timed

:retry / :retry This is my new input

:retry tells the system to rerun the most recent input on the state of the system a moment ago, and retry it. It should do exactly what it did before, but this time if you have turned on tracing it will trace it.

It performs this magic in stand-alone mode by copying the user's topic file into the TMP directory before each volley, so it can back up one time if needed. Because it operates from that tmp copy you can, if your log file is currently messy, merely erase all the contents of the USER folder before executing the revert and the log trace will only be from this input.

You can also put in different input using :retry. This is a fast way to try alternatives in a context and see what the system would do.

You can also designate a file you want loaded instead (if you have copied some previous user topic file out) by saying

:retry FILE filepath _This is my input_

:retry is expected to be used on a stand-alone system. NORMAL servers only allow users in the authorizedip.txt file to use debug commands.. If retry has been server enabled using the command line parameter "serverretry", thereafter each volley is tracked

:redo 5 This is new input

:redo is like :retry but takes a volley number and requires substitute input.

The command line parameter redo enables this command (otherwise becomes a :retry). This memorizes all inputs (and numbers the outputs with the current volley number). You can then back up to the input leading to any volley number by saying :redo n here is new input. Can chew up file space in TMP directory

:do stream

This execute sthe stream specified as though a rule has matched input and has this stream as its output section. E.g.

```
:do pos(noun word plural) will output "words".
:do $cs_token |= #STRICT_CASING will augment the user variable.
```

Sometimes you need the system to set up a situation (typically pronoun resolution).

```
:do hello world
```

will tell the system to generate hello world as its output.

In all cases the system will literally pretend a dummy user input of `:do`, incrementing the input count, and running the preprocess. Then, instead of running the main control topic, it merely executes the stream you gave it as though that were from some rule matched during main processing. The output generated is handled per normal and the system then runs the postprocess. The results are saved in the user's topic file.

:topics This is my sample input

This will display the topics whose keywords match the input, the score the topic gets, and the specific words that matched.

:say this is bot output

This will output whatever you wrote as though the chatbot said it. Useful for testing postprocessing code.

:silent

This toggles whether or not output is sent to the user. Useful when running regression tests.

:diff file1 file2 optional-separator

Reports on lines that are different between the two files. Useful when running regression tests. Often used with `:silent` and `:log` as well.

The system normally compares the lines of the two files, bypassing leading and trailing whitespace. If you provide an optional separator character, it will only compare the lines up to but not including that character, though it will display

the full lines when they fail to match. Aside from the usual echoing into the user's log file, the differences are also written to `LOGS/diff.txt`.

Other Debug Commands

`:show`

The `:show` choices are all toggles. Each call flips its state. However, you can supply a second argument to `:show`, which is a value and instead of toggling the flag will be set to that value.

`:show all`

toggles a system mode where it will not stop after it first finds an output. It will find everything that matches and shows all the outputs. It just doesn't proceed to do gambits. Since it is showing everything, it erases nothing. There is a system variable %all you can query to see if the all mode is turned on if you want some of your script to be unreactive when running all.

`:show input`

displays the things you send back into the system using the `^input` function.

`:show mark`

is not something you are likely to use but it displays in the log the propogation of marked items in your sentence. If you do `:echo` that stuff will also display on your screen.

`:show newline`

normally the log file transcodes all newline/return characters so that the log line of the bot output is all one line (so various tools work). But if you have an indented/newline oriented display you want to see in the logs, you can make this true. It does ruin the ability to use other tools like :regress, etc.

`:show number`

displays the current input number at the front of the bot's output. It is always shown in the log, but this shows it on-screen.

I use this before running a large regression test like `:source REGRESS/bigregress.txt` so I will know how far it has gotten while it's running.

`:show pos`

displays summary on the POS-tagging. Not useful for a user, but useful to me in debugging the engine itself.

`:show topic`

displays the current topic the bot is in, prefixed to its output.

`:show topics`

displays all the topics whose keywords matched the input.

`:show why`

this turns on `:why` for each volley so you always see the rule causing output before seeing the output.

`:log xxxx`

put the message you write directly into the log file. Useful for testers to send comments back to scriptors in the moment of some issue arising.

`:noreact`

toggles whether the system tries to respond to input.

`:testtopic ~topic sentence`

This will execute responder mode of the named topic on the sentence given, to see what would happen. It forces focus on that topic, so no other topic could intervene. In addition to showing you the output it generates, it shows you the values of all user variables it changes.

`:testpattern (...) sentence`

The system inputs the sentence and tests the pattern you provide against it. It tells you whether it matched or failed.

```
:testpattern ( it died ) Do you know if it died?
```

Some patterns require variables to be set up certain ways. You can perform assignments prior to the sentence.

```
:testpattern ($gender=male hit) $gender = male hit me
```

Typically you might use `:testpattern` to see if a subset of your pattern that fails works, trying to identify what has gone wrong. The corresponding thing for testing output is `:do`. You can also name an existing rule, rather than supply a pattern.

```
:testpattern ~aliens.ufo do you believe in aliens?
```

`:topicstats`

This walks all topics and computes how many rules of various kinds you have (e.g., how big is your system). You can also just name a topic or use a wildcard like `~do*` to see all topics starting with `~do`.

If you do :topicstats labels you just get the name of the topic and then 1 line for each rule label (indented to show rule hierarchy relationship). Simiarly :topicstats labels ~mytopic for just a single topic.

`:skip n`

The system will disable the next n gambits of the current topic, and tell you where you will end up next. Thereafter your next simple input like *ok* will execute that gambit, and the n previous will already have been used up.

Data Display Commands

The system is filled with data, some of which you might want to see from time to time.

`:pending`

Show the interesting topics list. Topics that have been entered recently but are not the current topic.

`:commands`

Displays available commands and a brief statement of purpose.

`:concepts word`

Lists all concepts that are affiliated with this word.

`:conceptlist {argument}`

List all concepts defined in the system or with an argument like `~w*`, show all concepts starting with `~w`

`:definition ^xxxx`

Shows the code of the user-defined macro named.

`:mixedcase`

Lists all words which have multiple case forms and where the words came from (dictionary, layer 0, layer 1). Helps you locate words where you inadvertently created a wrong case.

`:directories`

Lists the directories in use: the path to the exe file, the current working directory, and directory overrides used in embedded applications.

`:findwords word`

Given a word pattern, find all words in the dictionary that match that pattern. The pattern starts with a normal word character, after which you can intermix the wildcard * with other normal characters. For example,

- `slo*` finds *slothful*, *slower*, *sloshed*.
- `s*p*y` finds *supposedly*, *spendy*, and *surprizingly*.

`:functions`

Show all `^functions` defined by the system.

:macros

Show all ^macros defined by the user.

:memstats

Show memory use, number of words, number of facts, amount of text space used, number of buffers allocated.

:queries

Show all defined queries.

:variables {kind}

Rarely will the issue be that some variable of yours isn't correct. If you provide an argument, "system" restricts it to system variables and "user" restricts it to user variables, and "match" restricts it to all match variables and "bot" restricts it to bot variables.

:who

Show name of current user and current bot.

:nonset type set

find what words of the given part of speech are not encompassed by the named concept. This is a command to determine if some words are not covered by an ontology tree, and not used by normal scripters. E.g., `:nonset NOUN ~nounlist`.

:common dog cat

Given 2 or more words, this displays the concepts they share in common, most close first. This only applies to statically defined concepts, and not to dynamic engine concepts like parts-of-speech, role-in-sentence, `~number`, etc

`:word apple`

Given a word, this displays the dictionary entry for it as well some data up it's hierarchy. The word is case sensitive and if you want to check out a composite word, you need to use underscores instead of blanks. So `:word TV_star`.

Shows a limited number of facts involving the word, and an optional 2nd argument of a count of facts can limit or expand how many are shown.

`:fact apple`

Given a word, this displays facts which have that as one of the fields. The word is case sensitive and if you want to check out a composite word, you need to use underscores instead of blanks. So `:word TV_star`. If the word is a fact set id like @1, then all facts in that set are displayed.

Shows a limited number of facts involving the word, and an optional 2nd argument of a count of facts can limit or expand how many are shown.

`:dualupper`

Lists words that have multiple upper-case spellings. Ideally there would be only one such spelling.

`:userfacts`

This prints out the current facts stored with the user.

`:quotelines`

This takes a filename as argument, reads the lines and rewrites them to TMP/tmp.txt with double quotes around them. Useful for converting multiple word values into quoted ones for concepts.

`:allfacts`

This dumps a list of all the facts (including system facts) into the file `TMP/facts.txt`.

:facts meaning / :facts @n

This displays all facts for the given word or meaning. If you give a meaning (e.g., `sky~1`) then only facts involving that specific meaning are displayed. You can also give a fact like (subject verb object) and all facts containing that fact will be shown, but the fact can not contain any facts itself, it must be all simple fields.

If you use `:facts @2` then it will display all facts in that fact set. Bear in mind that normally factsets are NOT stored across volleys, so displaying a factset will likely show nothing unless you've executed ^save to protect it.

:up word

Shows the dictionary and concept hierarchy of things above the given word or concept.

:down word n

Shows the dictionary hierarchy below a word or, if the word is name of a concept, the members of the concept. Since displaying down can subsume a lot of entries, you can specify how many levels down to display (n). The default is 1.

:allmembers ~concept ~notconcept ~notconcept1 ...

Dumps the members of ~concept into TMP/tmp.txt, one per line, but excludes any that are member of the ~nonconcept sets.

:findwords pattern

This uses the pattern of characters and * to name words and phrases in the dictionary matching it. E.g.

```
:findwords *_executive
chief_executive
railroad_executive
:findwords *f_exe*
chief_executive
chief_executive_officer
Dancing_and_singing_are_my_idea_of_exercise.
```

`:overlap set1 set2`

This tests atomic members of set1 to see if they are also in set2, printing out the ones that are in both.

System Control Commands

`:build`

This compiles script into ready-to-use data in the TOPICS folder. You name a file to build. If the file name has a 0 at the end of it, it will build as level 0. Any other file name will build as level 1. You can build levels in any order or just update a single level.

A build file is named `filesxxx.txt` where the xxx part is what you specify to the build command. So `:build angela0` will use `filesAngela0.txt` to build level 0. A build file has as its content a list of file paths to read as the script source data. It may also have comment lines starting with `#`.

These paths are usually relative to the top level directory. E.g,

```
# ontology data
RAWDATA/ONTOLOGY/adverbhierarchy.top
RAWDATA/ONTOLOGY/adjectivehierarchy.top
RAWDATA/ONTOLOGY/prepositionhierarchy.top
```

Depending on what you put into it, a build file may build a single bot or lots of bots or a common set of data or whatever.

`:bot sue`

Change focus to conversing with the named bot (presuming you have such a bot). It resets the user back to complete new, flushing the users history, variables, etc.

`:language french`

Change current language of user to named language which must be in the language= command line parameter. Given no argument, it returns the current language.

`:reset`

Flush the current user's total history (erases the `USER/TOPIC` file) and reruns the bot definition macro, starting conversation from the beginning. Can be done from within a script without damaging the existing control flow or input sentence

but that means variables and facts not covered by the bot definition will remain untouched. For a complete reset see command line parameter `erasename`.

`:user username`

Change your login id. It will then prompt you for new input as that user and proceed from there, not starting a new conversation but merely continuing a prior one.

`:tsvsource filename conversationid message speaker count skip`

The system will switch to reading input from a delimited file describing conversations.

Conversationid is the field index where the id for login will be found. Message is the field index of the input being supplied. Speaker is field index of who is speaking (Professional or Customer).

Each successive line with the same conversation id continues the conversation. Either speaker can be doing the talking. When a new conversation id is read, the prior conversation ends and a new one begins. As the conversation ends, a topic you can define called ~cs_tsvsource which consists only of gambits. It is not harmful if you don't have this topic. It allows you to do any final processing of the conversation.

`:source REGRESS/bigregress.txt`

Switch the system to reading input from the named file. Good for regression testing.

The system normally prints out just the output, while the log file contains both the input and the output. You can say `:source filename echo` to have input echoed to the console.

If you say `:source filename internal` the system will echo the input, then echo the tokenized sentences it handled.

An input sentence that is merely `:quit` or `:exit` will stop source reading before the file ends, and return to normal source input.

:source can accept a log file and will execute the log entry data appropriate (and skip non-useful lines).

`:pos`

This is a subset of :prepare that just runs the POS-tagger parser on the input you supply. I use it to debug the system. It either is given a sentence or toggles a mode if not (just like `:prepare`). It also displays pronoun data gathered from the input.

`:tokenize`

This is a subset of :prepare that just runs tokenization and adjusts the input, but does not show POS-tagging and concept marking.

`:canonize`

This is a subset of :prepare that shows the canonical form of the input

`:logging`

Dynamically alter logging behavior from whatever current settings are.

```
:logging user {file,none,stdout,stderr,prelog}
:logging servertrace {on, off}
:logging server {file,none,stdout,stderr,prelog}
```

Serverlogging is a location for the log file. It does not matter if CS is acting as a server or not. Server log files go in LOGS/serverlogxxxx.txt where xxx is the port number.

Userlogging is stored in USERS folder, in files named by the bot and user. And userlogging is needed if you want to trace user processing, which goes into that user's file.

`:testpos`

This switches input to the named file (if not named defaults to `REGRESS/posttest.txt`) and running regression POS testing. If the result of processing an input deviates from that listed in the test file, the system presents this as an error.

`:verifysubstitutes`

This tests each substitution in the `LIVEDATA/substitutes` file to see if it does the expected thing.

:verifyspell

This tests each spelling in the `LIVEDATA/spellfix` file to see if it does the expected thing.

:verifypos

This tests pos regression data in REGRESS to see if it does the expected thing.

:fuse filename
This will read in the file named, replace new lines with spaces, and write to TMP/tmp.txt. This makes a multiple line file into a single file. This is handy if you are trying to create a large json structure to be passed as user input, which is easy to read in multiple lines, but then must be reformatted to single line for input.

:fuse filename

:restart

This will force the system to reload all its data files from disk (dictionary, topic data, live data) and then ask for your login. It's like starting the system from scratch, but it never stops execution. Good for revising a live server. But it does not return a message to the user, it restarts immediately. If you want to complete the current volley, say ":restart completevolley"

You may also say `:restart erase` to force the existing user to start with a new topic file (restarting a system might load new incompatible data if you've changed something).

You may also give up to 4 parameters exactly as you would on the command line to alter startup behavior. E.g., from script:

```
^eval(:restart erase Vserver=app.john.com/api/1.1 )
```

:restart redo_boot will, at end of volley, unwind the boot layer and reexecute boot functions to repopulate the server with new boot data visible to all bots and users.

:autoreply xxx

This command causes the system to talk to itself. As the user it always says whatever you put in xxx.

Debugging Function `^debug()`

As a last ditch, you can add this function call into a pattern or the output and it will call DebugCode in `functionExecute.cpp` so you know exactly where you are and can use a debugger to follow code thereafter if you can debug c code.

Logging Function `^log(...)`

This allows you to print something directly to the users log file. You can actually append to any file by putting at the front of your output the word FILE in capital letters followed by the name of the file. E.g.,

```
^log(FILE TMP/mylog.txt This is my log output.)
```

Logging appends to the file. If you want to clear it first, issue a log command like this:

```
^log(FILE TMP/mylog.txt NEW This is my log output)
```

The "new" tells it to initialize the file to empty. Additionally you can optimize log file behavior. If you expect to write to a file a lot during a volley (eg during :document mode), you can leave the file open by using

```
^log(OPEN TMP/mylog.txt This is my log output.)
```

which caches the file ptr. After which you can write with OPEN or FILE equivalently. To close the file use

```
^log(CLOSE TMP/mylog.txt)
```

You can log into the existing serverlog with ^log(SERVERLOG ...). You can log into the existing userlog with ^log(USERLOG ...). Actually getting stuff put into those files will depending on whether logging into those files is enabled.

Login after crash

If you want to repeat a crash and not go thru all the trouble to recreate the situation you can do the login differently. The old status of the system is still in the user's folder. Normally when you login, it picks that up, but begins a new conversation.

To resume the old conversation as though you had never left, login with `loginname:&`. If before you were user bruce, login as `bruce:&`. Now if you say what you said before, it should crash just the same. Not that that will do most of you any good, but it's handy if you can debug src code.

When all else fails

Usually you can email me for advice and solutions.

:sortconcept

This takes one or all concepts currently compiled, and can neaten them up in various ways.

`:sortconcept ~concept 1`

will take the named concept and write it out to `cset.txt` as a single line, with members sorted alphabetically.

`:sortconcept ~concept 0`

will take the named concept and write it out to `cset.txt` as a multiple tidy lines, with members sorted alphabetically.

`:sortconcept 1`

will take the all built concept and write them out to `concepts.top` as a single line, with members sorted alphabetically and concept declarations sorted alphabetically. You can use this output to replace your original scripted concepts.

`:sortconcept 0`

will take the all built concept and write them out to `concepts.top` as a multiple line, with members sorted alphabetically. If you want the concept declarations sorted alphabetically, first do `:sortconcept 1`, then take `concepts.top` and replace your original script concepts from it. Then do this and then replace your original script concepts from this.

:variablereference

List all user variable assignments in a bot and uses into TMP/reference.txt . See documentation in Finalize a bot

^testpattern and ^testoutput

There are special debugging abilities tied to ^testpattern.

A pattern that has %trace-on in it, turns on :trace all until either the pattern ends or %trace-off is found. This will output in the trace field of json return.

If the user message starts with `cheat cs info` then the system will output a newglobal $cs_info with the timestamps of the engine compilation, and the level 0 and level1 script.

If user input contains the serverauthorizationcode you supply in cs_initmore.txt, followed immediately by 2, then if there is a double quoted string as the next token, and the first token of that string is the label of the currently executing testpattern node, the rest of the string will be compiled, executed, and its output returned as a debug field. If no testpattern nodes match that name, then the first testoutput node that is invoked will execute the debug string. Eg.

```
    cs_initmore.txt:
serverauthcode=mycheat
    userinput:
I love mycheat2 "NODE4 :do test %bot me" in the morning.
```

The above should return a debug field with "test somebotname me" in it. If the ^testpattern NODE4 is never executed then the first ^testoutput node will execute this.

Whereas: `userinput: I love mycheat1 in the morning.` will execute a simple :trace pattern with output in trace field in the resulting object.