ChatScript Debugging Manual

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

Revision 8/17/13 cs3.53

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. Debugging mostly done by issuing commands to the engine, as opposed to chatting.

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

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

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.

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

Verification (:verify)

When I write a topic, before every rejoinder and every responder, I put a sample input comment. This has #! as a prefix. E.g.,

```
topic: ~mytopic (keyword)
t: This is my topic.
        #! who cares
        a: (who)  I care.

#! do you love me
?: (do you love)  Yes.

#! I hate food
s: (I * ~hate * food) Too bad.
```

This serves two functions. First, it makes it easy to read a topic-- you don't have to interpret pattern code to see what is intended. Second, it allows the system to verify your code in various ways. It is the “unit test” of a rule. If you've annotated your topics in this way, you can issue

:verify
:verify ~topicname
:verify keyword
:verify pattern ~topic
:verify blocking

There are several verifications the system can do. The first command verifies all topics in all ways. The second restricts itself to the specific topic listed in all ways. The third verifies keywords of all topics. The fourth verifies just the patterns of rules in the given topic. The fifth verifies blocking of all topics.

:verify keyword: For responders, does the sample input have any keywords from the topic. If not, there may be an issue. Maybe the sample input has some obvious topic-related word in it, which should be added to the keywords list. Maybe it's a responder you only want matching if the user is in the current topic. E.g.,

```
#! Do you like swirl?
?: ( swirl) I love raspberry swirl
```

can match inside an ice cream topic but you don't want it to react to *Does her dress swirl?*.

Or maybe the sample input has no keywords but you do want it findable from outside (E.g., an idiom), so you have to make it happen. When a responder fails this test, you have to either add a keyword to the topic, revise the sample input, add an idiom table entry, or tell the system to ignore keyword testing on that input.

You can suppress keyword testing by augmenting the comment on the sample input with !K.

```
#!!K this is input that does not get keyword tested.
```

:verify pattern: For responders and rejoinders, the system takes the sample input and tests to see if the pattern of the following rule actually would match the given input. Failing to match means the rule is either not written correctly for what you want it to do or you wrote a bad sample input and need to change it.

Rules that need variables set certain ways can do variable assigns ($ or _) at the end of the comment. You can also have more than one verification input line before a rule.

```
#! I am male  $gender = male
#! I hate males $gender = male
s: ($gender=male I * male) You are not my type
```

If you want to suppress testing, add !P to the comment.

```
#!!P This doesn't get pattern testing.
```

If you want to suppress pattern and keyword testing, just use K and P in either order:

```
#!!KP this gets neither testing.
```

You can also test that the input does not match the pattern by using #!!R instead of #!, though unless you were writing engine diagnostic tests this would be worthless to you.

:verify blocking: Even if you can get into the topic from the outside and the pattern matches, perhaps some earlier rule in the topic can match and take control instead. This is

called blocking. One normally tries to place more specific responders and rejoinders before more general ones. The below illustrates blocking for *are your parents alive?* The sentence will match the first rule before it ever reaches the second one.

```
#! do you love your parents
?: ( << you parent >>) I love my parents       *** this rule triggers by mistake
#! are your parents alive
?: ( << you parent alive >>) They are still living
```

The above can be fixed by reordering, but sometimes the fix is to clarify the pattern of the earlier rule.

```
#! do you love your parents
?: ( ![alive living dead] << you parent >>) I love my parents
#! are your parents alive
?: ( << you parent alive >>) They are still living
```

Sometimes you intend blocking to happen and you just tell the system not to worry about it using !B.

```
#! do you enjoy salmon?
?: ( << you ~like salmon >>) I love salmon
#!!B do you relish salmon?
?: (<< you ~like salmon >>) I already told you I did.
```

The blocking test presumes you are within the topic and says nothing about whether the rule could be reached if you were outside the topic. That's the job of the keyword test. And it only looks at you sample input. Interpreting your pattern can be way too difficult.

If :trace has been set non-zero, then tracing will be turned off during verification, but any rules that fail pattern verification will be immediately be rerun with tracing on so you can see why it failed.

Passing verification means that each topic, in isolation, is plausibly scripted. Other topics might still grab control, but that's a different issue. So, you should verify first and live test second. But things can still go wrong. Or maybe you need to debug to find out why :verify pattern failed.

:topicinfo ~topic how

This displays all sorts of information about a topic including its keywords, how they overlap with other topics, what rules exist and whether they are erased or not. You either name the topic or you can just you ~, which means the current rejoinder topic (if there is one).

If how is omitted, you get everything. You can restrict things with a collection of how keywords. These include “keys”, to display the keywords, “overlap” to display the overlap with other topics' keywords, “gambits”, “rejoinders”, “responders” or “all” to

limit rules to some of those, “used” and “available” to see only those rules meeting that criteria.

Keyword overlap is particularly interesting. As you assign keywords to topics, at times you will probably get excessive. Some topics will share keywords with other topics. For some things, this is reasonable. “quark” is a fine keyword for a topic on cheese and one on astronomy. But odds are “family” is not a great keyword for a topic on money. Often an extraneous keyword won't really matter, but if the system is looking for a topic to gambit based on “family”, you really don't want it distracted by a faulty reference to money. That is, you want to know what keywords are shared across topics and then you can decide if that's appropriate.

Validation

Having built a functioning chatbot with lots of topics, I like to insure I don't damage old material in the future, so I write a list of test questions or statements for each topic. I refer to this as validation. I run the chatbot against this file using :source. I take the resulting log file, move it to LOGS directory and type :trim 2 . This generates a file TMP/tmp.txt which lists three things: the topic an answer came from, the sample input, and the actual output. I save this file away somewhere.

Later on, I can show the chatbot is undamaged by :source of the validation input file, doing a :trim 2 on the moved log file, and then doing a “diff” using a diff tool on the new output of trim vs the old output. If I see changes, I have to account for them either by fixing the chatbot, or replacing the standard output base with the new result.

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.

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.

:why

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. So I enable some trace (being lazy I typically do :trace all or :trace ~topicname where the rule I'm interested in is in ~topicname) and then I say

: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.

:do stream – execute the 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 $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.

The Debugger

You can run an embedded debugger and step thru execution yourself, setting breakpoints, displaying and modifying values, and see what goes wrong.

:debug what is your name

The above launches the debugger and specifies to debug the listed input. A special form of the command

:debug :retry

combines debugging with the :retry command. The system's state is restored to before the most recent input, that input is then used as the input for :debug.

The debugger will normally stop at the beginning of each of the three phases (preprocess, main, postprocess)/ You may not have each of these defined to topics, in which case you will see fewer phases debugged. You give commands, some of which eventually tell the debugger to resume executing, perhaps displaying stuff, until it reaches a next pausing place or completes.

Data put out by the system will be about topics (entering and exiting) and rules. The data will be indented to show its nesting relationship to other data. When the system is paused, awaiting commands, it will display a "?:" prompt. Many commands only require a single letter. Some require an entire line of input followed by the enter key. You can get a list of commands by typing *h*, which is the help command. Normally you continue execution using one of the arrow keys.

Conceptually the debuggable items are *topics*, *rules,* and *actions.* Rules are all of the gambits, responders, and rejoinders. Actions are the things you can do in the output section of a rule.

You can choose to step over an item, in which case it will complete its execution and the system moves on to the next item.

You can choose to step into an item. This will look at things that happen inside the item. For a topic, the next item layer in are rules. For rules, the next item layer in are actions. Some actions also have a next layer. Actions *Respond()* and *Gambit()* take you to a topic item. Action *Reuse(), Refine(),* and *Loop()* take you to a subcollection of rules. *If* and *Loop* have actions inside of them. The subcollections themselves are items.

You can choose to step out of an item. This means completing execution of everything inside, and leaving the item, returning to the item that invoked this item and moving on to the next. i.e., you have now completed *stepping over* the original item.

Sometimes *step in, step over,* and *step out* have the same effect. For example, if you are at a rule that will not match, then stepping in does nothing and the rule is complete. It is the same as step over. If that was the last rule of a topic, then the only place to go is to

leave the topic, so it becomes the same as step out. Similar behaviors can happen when you try to step in on an action that does nothing special. The action merely completes, which is the same as step over. And if it's the last action of a rule, then the rule is complete and you have "stepped out".

Right-arrow - "step over" - execute the current data item in its entirety. For a topic, this means all rules of the topic and everything they call. The topic will complete. For a rule, this means completing the rule and moving on to the next rule. For an action this means performing whatever it requests, including doing an nested topic calls, etc.

Down-arrow - "step in" - execute the next lower level. If you are in a topic, the next level down is a rule of the topic. For a rule, the next level down is its actions. For an action, some of them can call topics or subcollections of rules.

Left-arrow - "run to completion" - run until finished. Any traces and breakpoints will be obeyed.

Up-arrow - "step out" - "step out" means finish up all the work at this level and return to the higher level. A topic finishes all its rules and exits. A rule finishes all its actions and moves on to the next rule. An action finishes its sub-behaviors and moves on to the next action. A subcollection like *loop* finishes the loop and moves on to the next rule.

h (help) – displays the list of commands.

x (exit) - stop debugging and complete processing the input. You might do this if you were actually finished debugging. Or maybe you skimmed over the actual important area of debugging and want to try again more carefully (in which case :debug :retry should be your next command).

b xxx yyy ENTER (set breaks) – b followed by a list and then the enter key will set breakpoints. An item in the list can be a topic name (topic breakpoint) or a rule tag or rule label (rule breakpoint).

l (list breaks) – this will print out the list of current breaks you have set.

g (optional breakpoint) – go until - this is like a "step out" command requiring ENTER at the end, that will also stop if it reaches the named topic or rule (naming the break is optional).

d xxx yyy ENTER (delete breaks) – d followed by a list and then the enter key will remove those breakpoints.

m – "run til match" resume executing, displaying topic entry/exit, but stop at the next matching rule. Leaping from matching rule to matching rule is the fast overview of what happens, not seeing all the failed rules.

t (trace topics) – Toggle a flag to trace entry and exit of all topics.

r (trace rules) – Toggle a flag to trace entry into a rule.

The data lines:

at ~topicname - displayed on entering a topic-related

exit ~topicname result: OK responses: 0 – displayed on exiting a topic, showing return status and current number of user outputs generated.

- *~topicname.3.0 u: (....) ...* - displays a rule to be executed. The – means it fails to match, a + means it would match and go process its output. The rule tag is displayed, followed by rule type and some of the actual rule code. When you are single stepping thru a topic, the rule is printed out before execution happens, so you can see what *will* or *will not* match, before you tell it to proceed. This involves pre-trying the match, so if your match actually had code that modified the world (most don't), things might go differently when the rule actually executes. An instance of that would be a pattern based on matching a value from the %random generator, which would change the value.

While you are paused in the debugger, you can execute many of the :commands, particularly the *:do* command. This will allow you to inspect individual variable values, and even change them. You can also execute arbitrary functions with *:do*, but understand that some things you execute might change the data your pending execution will depend upon. You can, for example, use the debugger to get to a certain spot, then turn on tracing with :trace, then resume execution.

Other Debug Commands

:show

The :show choices are all toggles. Each call flips its state.

: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 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.

:topicstats – walks all topics and computes how many rules of various kinds you have (e.g., how big is your system).

: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.

:trace all
:trace none

The ultimate debugging command dumps a trace of everything that happens during execution onto 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.

:trace ~education **-** this enables tracing for this topic and all the topics it calls. Call it again to restore to normal.

:trace ^myfunction – this enables tracing for the function. Call it again to restore to normal.

: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.

:trace ~education.school – this traces all top-level rules in ~education that you named *school* (and its rejoinders and anything it calls. Call it again to turn off the trace.

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:

t*opic: foo […]*
.....
:trace all

Tracing is also good in conjunction with some other commands that give you a restricted view.

Data Display Commands

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

:interesting

Show the interesting topics list.

:commands

Displays available commands and a brief statement of purpose.

:definition ^xxxx

Shows the code of the user-defined macro named.

: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.

:variables {kind}

Rarely will the issue be that some variable of yours isn't correct. But you can show the values of all user $variables and all system %variables. If you provide an argument, “system” restricts it to system variables and “user” restricts it to user 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 .

: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 .

:userfacts

This prints out the current facts stored with the user.

:allfacts

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

: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.

: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).

:reset

Flush the current user's total history (erases the USER/ topic file), starting conversation from the beginning.

: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.

: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.

: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.

: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.

: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. You can optionally name a language to restart under, allowing you to switch dictionary bases.

:autoreply why
:autoreply ok

These commands cause the system to talk to itself. As the user it always says why or OK. Not something you are likely to use.

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.)

When all else fails

Usually you can email me for advice and solutions.