ChatScript System Variables and Engine-defined Concepts and Parameters

Revision 1/7/2017 cs7.1

Engine-defined concepts

In addition to concepts defined in script files, the system automatically defines a bunch of dictionary-based sets as well as dynamically computed concept members.

set	description
`~web_url`	word is a web url
`~email_url`	word is an email address
`~kindergarten`	word learned early in life
`~grade1_2`	word learned in these grades
`~grade3_4`	word learned in these grades

set	description
`~grade_5-6`	word learned in these grades. Unmarked words are learned even later
`~utf8`	word has nonascii characters
`~daynumber`	word could be a number of a day in a month
`~yearnumber`	word could be the number of a recent year
`~dateinfo`	phrase is month day year of some kind
`~kelvin`	temperature marker
`~celcius`	temperature marker
`~fahrenheit`	temperature marker

set	description
~twitter_name	twitter user name
~hashtag_label	twitter topic reference

Interjections, "discourse acts", and concept sets

Some words and phrases have interpretations based on whether they are at sentence start or not. E.g., *good day, mate* and *It is a good day* are different for *good day*.

Likewise sure and I am sure are different. Words that have a different meaning at the start of a sentence are commonly called interjections.

In ChatScript these are defined by the `livedata/interjections.txt` file. In addition, the file augments this concept with "discourse acts", phrases that are like an interjection. All interjections and discourse acts map to concept sets, which come thru as the user input instead of what they wrote. For example yes and sure and of course are all treated as meaning the discourse act of agreement in the interjections file. So you don't see yes, I will go coming out of the engine.

The interjections file will remap that to the sentence ~yes, breaking off that into its own sentence, followed by I will go as a new sentence.

These generic interjections (which are open to author control via interjections.txt) are: `~yes`,`~no`,`~emomaybe`,`~emohello`,`~emogoodbye`,`~emohowzit`,`~emothanks`, `~emolaugh`,`~emohappy`,`~emosad`,`~emosurprise`, `~emomisunderstand`, `~emoskeptic`,`~emoignorance`,`~emobeg`, `~emobored`, `~emopain`,`~emoangry`, `~emocurse`,`~emodisgust`,`~emoprotest`, `~emoapology`,`~emomutual`

Because all interjections at the start of a sentence are broken off into their own sentence, this kind of pattern does not work:

```
u: (~yes _*)
```

You cannot capture the rest of the sentence here, because it will be part of the next sentence instead. This means interjections act somewhat differently from other concepts.

If you use a word in a pattern which may get remapped on input, the script compiler will issue a warning. Likely you should use the remapped name instead.

The following concepts are triggered by exactly repeating either the chatbot or oneself (to a repeat count of how often repeated). Repeats are within a re-

cency window of about 20 volleys. ~repeatme, ~repeatinput1, ~repeatinput2, ~repeatinput3, ~repeatinput4, ~repeatinput5, ~repeatinput6,

POS (Part of Speech) Tags

Words will have pos-tags attached, specifiifying both generic and specific tag attributes, eg., ~noun and ~noun_singular.

Genric Specifics

~noun, ~noun_singular, ~noun_plural, ~noun_proper_singular, ~noun_proper_plural, ~noun_gerund, ~noun_number, ~noun_infinitive, ~noun_omitted_adjective, ~verb, ~verb_present, ~verb_present_3ps, ~verb_infinitive, ~verb_present_participle, ~verb_past, ~verb_past_participle, ~aux_verb, ~aux_verb_present, ~aux_verb_past, ~aux_verb_future (~aux_verb_tenses), ~aux_be, ~aux_have, ~aux_do

Auxilliary verbs are segmented into normal ones and special ones. Normal ones give their tense directly. Special ones give their root word. The tense of the be/have/do verbs can be had via ^properties() and testing for verb tenses

~adjective, ~adjective_normal, ~adjective_number, ~adjective_noun, ~adjective_participle

Adjectives in comparative form will also have ~more_form or ~most_form. ~adverb, ~adverb_normal

Adverbs in comparative form will also have ~more_form or ~most_form. ~pronoun, ~pronoun_subject, ~pronoun_object, ~conjunction_bits, ~conjunction_coordinate , ~conjunction_subordinate, ~determiner_bits, ~determiner, ~pronoun_possessive, ~predeterminer, ~possessive (covers 'and 's at end of word), ~to_infinitive ("to" when used before a noun infinitive), ~preposition, ~particle (free-floating preposition tied to idiomatic verb), ~comma, ~quote (covers' and " when not embedded in a word), ~paren (covers opening and closing parens), ~foreign_word (some unknown word), ~there_existential (the word there used existentially),

In addition to normal generic kinds of pos tags, words which are serving a pos-tag role different from their putative word type are marked as members of the major tag they act as part of. E.g,

~noun_gerund – verb used as a ~noun ~noun_infinitive – verb used as a ~noun ~noun_omitted_adjective – an adjective used as a collective noun (eg the beautiful are kind)

~adjectival_noun (noun used as adjective like bank "bank teller") ~adjective_participle (verb participle used as an adjective)

For ~noun_gerund in *I like swimming* the verb gerund *swimming* is treated as a noun (hence called noun-gerund) but retains verb sense when matching keywords tagged with part-of-speech (i.e., it would match swim~v as well as swim~n).

~number is not a part of speech, but is comprise of ~noun_number (a normal number value like *17* or *seventeen*) and ~adjective_number (also a normal numeral value and also ~placenumber) like first. Additionally, there is ~integer, ~float, ~positiveinteger, and ~negativeinteger.

To can be a preposition or it can be special. When used in the infinitive phrase To go, it is marked ~to_infinitive and is followed by ~noun_infinitive.

~verb_infinitive refers to a match on the infinitive form of the verb (I hear John sing or I will sing).

~There_existential refers to the use of where not involving location, meaning the existence of, as in There is no future.

~Particle refers to a preposition piece of a compound verb idiom which allows being separated from the verb. If you say *I will call off the meeting*, call_off is the composite verb and is a single token. But if you split it as in *I will call the meeting off*, then there are two tokens. The original form of the verb will be call and the canonical form of the verb will be call_off, while the free-standing off will be labeled ~particle.

~verb_present will be used for normal present verbs not in third person singular like I walk and ~verb_present_3ps will be used for things like he walks

~possesive refers to ‘s and ‘ that indicate possession, while possessive pronouns get their own labeling ~pronoun_possessive.

~pronoun_subject is a pronoun used as a subject (like he) while pronoun_object refers to objective form like (him)

Individual words serve roles in the parse of a sentence, which are retrievable. These include:

~mainsubject, ~mainverb, ~mainindirect, ~maindirect, ~subject2, ~verb2, ~indirectobject2, ~object2, ~subject_complement – (adjective object of sentence involving linking verb), ~object_complement – (2ndary noun or infinitive verb filling modifying mainobject or object2), ~conjunct_noun, ~conjunct_verb, ~conjunct_adjective, ~conjunct_adverb ~conjunct_phrase, ~conjunct_clause, ~conjunct_sentence, ~postnominalAdjective - adjective occuring AFTER the noun it modified, ~reflexive - (reflexive pronouns), ~not, ~address – noun used as addressee of sentence, ~appositive – noun restating and modifying prior noun, ~absolutephrase – special phrase describing whole sentence, ~omittedtimeprep – modified time word used as phrase but lacking preposition (Next tuesday I will go), ~phrase – a prepositional phrase start (except, ~clause – a subordinate clause start, ~verbal – a verb phrase.

System Variables

The system has some predefined variables which you can generally test and use but not normally assign to. These all begin with `%` . Ones that are reasonable to set are written in bold underline. Boolean values are always `1` or `null` on returns. `1` or `0` if you are setting them.

Date & Time & Numbers

variable	description
`%date`	one or two digit day of the month
`%day`	Sunday, etc
`%daynumber`	0-6 where 0 = Sunday
`%fulltime`	seconds representing the current time and date (Unix epoch time)
`%timenumbers`	completely consistent full time info in numbers that you can do `_0 = ^burst(%timenumbers)` to get `_0` =seconds (2digit) `_1`=minutes (2digit) `_2`=hours (2digit) `_3`=dayinweek(0-6 Sunday=0) `_4`=dateinmonth (1-31) `_5`=month(0-11 January=0) `_6`=year. You need to get it simultaneously if you want to do accurate things with current time, since retrieving %hour %minute separately allows time to change between calls
`%leapyear`	boolean if current year is a leap year
`%daylightsavings`	boolean if current within daylight savings
`%minute`	0-59
`%month`	1-12 (January = 1)
`%monthname`	January, etc
`%second`	0-59
`%volleytime`	number of seconds of computation since volley input started
`%time`	hh:mm in military 24-hour time
`%week`	1-5 (week of the month)
`%year`	e.g., 2011
`%rand`	get a random number from 1 to 100 inclusive

Time and date information are normally local, relative to the system clock of the machine CS is running on. See $cs_utcoffset for adjusting time based on relationship to utc (e.g your server is in Virginia and you are in Colorado).

User Input

variable	description
`%bot`	current bot responding
`%revisedinput`	Boolean is current input from ^input not direct from user
`%command`	Boolean was the user input a command
`%foreign`	Boolean is bulk of the sentence composed of foreign words
`%impliedyou`	Boolean was the user input having you as implied subject

variable	description
%input	the count of the number of volleys this user has made ever
%ip	ip address supplied
%length	the length in tokens of the current sentence
%more	Boolean is there another sentence after this
%morequestion	Boolean is there a ? or question word in the pending sentences

variable	description
`%originalinput`	all sentences user passed into volley, before adjusted in any way except OOB data is stripped off
`%originalsentence`	current sentence after tokenization but before any adjustments
`%parsed`	Boolean was current input parsed successfully
`%question`	Boolean was the user input a question – same as ? in a pattern

variable	description
%quotation	Boolean is current input a quotation
%sentence	Boolean does it seem like a sentence (subject/verb or command)
%tense	past , present, or future simple tense (present perfect is a past tense)
%user	user login name supplied
%userfirstline	value of %input that is at the start of this conversation start

variable	description
%userinput	Boolean is the current input from the user (vs the chatbot)
%voice	active or passive on current input

Chatbot Output

variable	description
%inputrejoinder	index of any pending rejoinder for input or 0 if none
%lastoutput	the text of the last generated response for the current volley
%lastquestion	Boolean did last output end in a ?

variable	description
%outputrejunder	number if system set a rejoinder for its current output or 0
%response	number of responses that have been generated for this sentence

System variables

variable	description
%all	Boolean is the :all flag on? (:all to set)
%document	Boolean is :document running
%fact	Numeric value most recent fact id

variable	description
%freetext	kb of available text space
%freedict	number of unused dictionary words
%freefact	number of unused facts
%maxmatchvariables	highest number of _match variables, currently 20
%maxfactsets	highest number of @fact-sets, currently 20
%host	name of the current host machine or "local"
%regression	Boolean is the regression flag on

variable	description
`%server`	Boolean is the system running in server mode
`%rule`	get a tag to the current executing rule. Can be used in place of a label

variable	description
`%topic`	name of the current "real" topic . if control is currently in a topic or called from a topic which is not system or nostay, then that is the topic. Otherwise the most recent pending topic is found
`%actualtopic`	literally the current topic being processed (system or not)

variable	description
%trace	Numeric value of the trace flag (:trace to set)
%httpresponse	return code of most recent ^jsonopen call
%pid	Linux process id or 0 for other systems
%restart	You can set and retrieve a value here across a system restart.

Build data+

variable	description
%dict	date/time the dictionary was built
%engine	date/time the engine was compiled
%os	os invovled (linux windows mac ios)
%script	date/time build1 was compiled
%version	engine version number

You actually can assign to any of them. This will override them and make them return what you tell them to and is a particularly BAD thing to do if this is running on a server since it affects all users (unless you reset the variable at the

end of the volley. Assigning a period to a variable resets it). Typically one does this as a temporary assignment in a #! comment line to set up conditions for testing using :verify. Making them return a new value is NOT the same thing as making the engine have a different value. Unless the variable is marked as settable, setting a value affects only the value returned by a future call to the system variable. It does not change engine values the variable is meant to reflect.

Control Over Input

The system can do a number of standard processing on user input, including spell correction, proper-name merging, expanding contractions etc. This is managed by setting the user variable `$cs_token`.

The default one that comes with Harry is:

```
$cs_token = #DO_INTERJECTION_SPLITTING |
            #DO_SUBSTITUTE_SYSTEM |
            #DO_NUMBER_MERGE |
            #DO_PROPERNAME_MERGE |
            #DO_SPELLCHECK |
            #DO_PARSE
```

The #signals a named constant from the dictionarySystem.h file. One can set the following:

These enable various LIVEDATA files to perform substitutions on input:

flag	description
#DO_ESSENTIALS	ENTIALS LIVE-DATA/systemessentials which mostly strips off trailing punctu-ation and sets corre-spond-ing flags instead

flag	description
#DO_SUBSTITUTES	perform LIVEDATA/substitutes
#DO_CONTRACTIONS	perform LIVE-DATA/contractions, expanding contractions
#DO_INTERJECTIONS	perform LIVE-DATA/interjections, changing phrases to interjections
#DO_BRITISH	perform LIVE-DATA/british, respelling brit words to American
#DO_SPELLING	performs the LIVE-DATA/spelling file (manual spell correction)
#DO_TEXTING	performs the LIVE-DATA/texting file (expand texting notation)

flag	description
`#DO_SUBSTITUTE_SYSTEM`	substitute LIVE-DATA file expansions
`#DO_INTERJECTION_SPLITTING`	break off leading interjections into own sentence
`#$DO_NUMBER_MERGE`	merge multiple word numbers into one (*four and twenty*)
`#$DO_PROPERNAME_MERGE`	merge multiple proper name into one (_George Harrison)
`#DO_DATE_MERGE`	merge month day and/or year sequences (*January 2, 1993*)

flag	description
`#JSON_DIRECT_FROM_OOB`	Direct the tokenizer to directly process OOB data. See ^json-parse in JSON manual.

If any of the above items affect the input, they will be echoed as values into %tokenFlags so you can detect they happened. The next changes do not echo into %tokenFlags and relate to grammar of input:

flag	description
`DO_POSTAG`	Allow pos-tagging (labels like ~noun ~verb become marked)
`DO_PARSE`	Allow parser (labels for word roles like ~main_subject)

flag	description
DO_CONDITIONAL_POSTAG	perform pos-tagging only if all words are known. Avoids wasting time on foreign sen-tences in particular
NO_ERASE	where a substi-tution would delete a word entirely as junk, don't

flag	description
`DO_SPLIT_UNDERSCORES`	Tapped after all other input tokenization and adjustments except number merge, and separates words that have been conjoined either because the dictionary has them (*credit_card*) or because they were merged by proper name merging, or by substitution. The result is only words without underscores (excluding number words like *five_thousand_and_four*)

flag	description
MARK_LOWER	If a word is considered a proper name in CS and is marked as an upper case word, this will force it to perform any markings for its lower case form as well. Sometimes users type stuff in upper case that really should be lower

Normally the system tries to outguess the user, who cannot be trusted to use correct punctuation or casing or spelling. These block that:

flag	description
STRICT_CASING	assume for 1st word of a sentence, assume user uses correct casing on words
NO_INFER_QUESTION	the system will not try to set the QUESTION-MARK flag if the user didn't input a ? and the structure of the input looks like a question
DO_SPELLCHECK	perform internal spell checking

flag	description
ONLY_LOWERCASE	Expect input (except "I") to be lower case, refuse to recognize upper-case forms of anything
NO_IMPERATIVE	
NO_WITHIN	
NO_SENTENCE_END	

Normally the tokenizer breaks apart some kinds of sentences into two. These prevent that:

flag	description
NO_COLON_END	Don't break apart a sentence after a colon
NO_SEMICOLON	Don't break apart a sentence after a semicolon

flag	description
`UNTOUCHED_INPUT`	this alone, will tokenize only on spaces, leaving everything but spacing untouched

flag	description
LEAVE_QUOTE	if input is found withing " " it will become a single token exactly as it is seen. W/o Leave_Quote, it is converted into a word without quotes and using underscores instead of spaces. So “My Fair Lady” becomes My_Fair_Lady, which would match a movie title if you had one, unlike *My Fair Lady* becoming the resulting token and unrecognized

flag	description

Note, you can change `$cs_token` on the fly and force input to be reanalyzed via `^retry(SENTENCE)`. I do this when I detect the user is trying to give his name, and many foreign names might be spell-corrected into something wrong and the user is unlikely to misspell his own name. Just remember to reset `$cs_token` back to normal after you are done. Here is one such way, assuming `$stdtoken` is set to your normal tokenflags in your bot definition outputmacro:

```
#! my name is Rogr
s: (name is _*)

    if ($cs_token == $stdtoken)
        {
        $cs_token = #DO_INTERJECTION_SPLITTING |
                    #DO_SUBSTITUTE_SYSTEM | #DO_NUMBER_MERGE |
                    #DO_PARSE
        retry(SENTENCE)
        }
    _0 is the name.
    $cs_token = $stdtoken
```

If you type *my name is Rogr* into a topic with this, the original input is spell-corrected to *my name is Roger*, but this will change the `$cs_token` over to one without spell correction and redo the sentence, which will now come back with *my name is Rogr* and be echoed correctly, and `$cs_token reset`. That's assuming nothing else would run differently and trap the response elsewhere. If you were worried about that, it would be possible for the script to save where it is using `^getrule(tag)` and modify your control script to return immediate control to here after input processing if you had changed `$cs_token`.

Private Substitutions

While in general, substitutions are defined in the LIVEDATA folder, you can define private substitutions for your specific bot using the scripting language. You can say

```
replace: xxx yyyyy
```

which defines a substitution just like a livedata substitution file. It actually creates a substitution file called `private0.txt` or `private1.txt` in your TOPIC folder. Even then, those substitutions will not be enacted unless you explicitly add to the $cs_token value #DO_PRIVATE, eg

```
$cs_token = #DO_INTERJECTION_SPLITTING |
            #DO_SUBSTITUTE_SYSTEM |
```

```
#DO_NUMBER_MERGE |
#DO_PROPERNAME_MERGE |
#DO_SPELLCHECK |
#DO_PARSE |
#DO_PRIVATE
```

Similarly while canonical values of words can be defined in `LIVEDATA/SYSTEM/canonical.txt`, you can define private canonical values for your bots by using the scripting language. You can say:

```
canon: oh 0 faster fast
```

which defines new canonical values for things and creates a file canon0.txt or canon1.txt in your TOPIC folder. If you want to set a canonical pair from a table during compilation, you can use a function to do the same thing (but only 1 pair at a time).

```
^canon(word canonicalform)
```

Interchange Variables

The following variables can be defined in a script and the engine will react to their contents.

interchange variable	description
`$cs_token`	described extensively above

interchange variable	description
$cs_response	controls automatic handling of outputs to user. By default it consists of $cs_response = #Response_upperstart | #response_removespacebeforecomma | #response_alterunderscores | #response_removetilde If you want none of theses, use $cs_response = 0 (all flags turned off). See ^print for explanation of flags. #response_noconvertspecial – leave escaped n r and t alone in output and ^log, #response_upperstart – makes the first letter of an output sentence

interchange variable	description
`$cs_jsontimeout`	seconds before JsonOpen declares a time out failure. If unspecified the default is 300
`$cs_crashmsg`	in server mode, what to say if the server crashes and we return a message to the user. By default the message is *Hey, sorry. I forgot what I was thinking about.*
`$cs_abstract`	used with :abstract

interchange variable	description
`$cs_looplimit`	loop() defaults to 1000 iterations before stopping. You can change this default with this

interchange variable	description
`$cs_trace`	if this variable is defined, then whenever the user's volley is finished, the value of this variable is set to that of :trace and :trace is cleared to 0, but when the user is read back in, the :trace is set to this value. For a server, this means you can perform tracing on a user w/o making all user transactions dump trace data

interchange variable	description
`$cs_control_pre`	name of topic to run in gambit mode on pre-pass, set by author. Runs before any sentences of the input volley are analyzed. Good for setting up initial values
`$cs_usermessagelimit`	max number of message pairs (user input & bot output) saved in topic file

interchange variable	description
`$cs_externaltag`	name of a topic to use to replace existing internal English pos-parser. See bottom of ChatScript PosParser manual for details

interchange variable	description
`$cs_prepass`	name of a topic to run in responder mode on main volleys, which runs before $cs_control_main and after all of the above and pos-parsing is done. Used to amend prepa-ration data coming from the engine. You can use it to add your own spin on input process-ing before going to your main control. I use it to, for exam-ple, label com-mands as ques-tions, stan-

interchange variable	description
`$cs_control_main`	name of topic to run in responder mode on main volleys, set by author
`$cs_control_post`	name of topic to run in gambit mode on postpass, set by author
`$botprompt`	message for console window to label bot output
`$userprompt`	message for console window to label user input line
`$cs_crashmsg`	message to use if a server crash occurs

interchange variable	description
`$cs_language`	if spanish, will adjust spell checking for spanish colloquial
`$cs_token`	bits controlling how the tokenizer works. By default when null, you get all bits assumed on. The possible values are in src/dictionarySystem.h (hunt for $token) and you put a # in front of them to generate that named numeric constant

interchange variable	description
`$cs_abstract`	topic used by :abstract to display facts if you want them displayed
`$cs_prepass`	topic used between parsing and running user control script. Useful to supplement parsing, setting the question value, and revising input idioms

interchange variable	description
`$cs_wildcardseparator`	when a match variable covers multiple words, what should separate them- by default it's a space, but underscore is handy too. Initial system character is space, creating fidelity with what was typed. Useful if _ can be recognized in input (web addresses). Changing to _ is consistent with multiword representation and keyword recogni-

interchange variable	description
`$cs_userfactlimit`	how many of the most recent permanent facts created by the script in response to user inputs are kept for each user. Std default is 100
`$cs_response`	controls some characteristics of how responses are formatted
`$cs_randIndex`	the random seed for this volley

interchange variable	description
`$cs_utcoffset`	if defined, then %time returns current utc time + time-zone offset. The offset is usually a simple number, meaning hours, and can have + or – in front of it. It can also be a normal time reference like 02:30 which means plus 2 hours and 30 minutes beyond utc, or -01:30:20 which means 1 hour, 30 minutes, and 20 seconds before utc (as if anyone would

interchange variable	description
`$$db_error`	error message from a postgres failure
	$$findtext_start - ^findtext return the end normally, this is where it puts the start
`$$tcpopen_error`	error message from a tcpopen error
`$$document`	name of the document being read in document mode
`$cs_randindex`	current value of the random generator value

interchange variable	description
`$cs_bot`	name of the bot currently in use
`$cs_login`	login name of the user
`$$csmatch_start`	start of found words from ^match
`$$csmatch_end`	end of found words from ^match

interchange variable	description
`cs_factowner`	when non-zero creates facts restricted by this bit-mask so facts created by other masks cannot be seen. allows you to separate facts per bot in a multi-bot environment. During compilation if this is set by a bot: command, then functions created and facts created by tables will be restricted to that owner.

interchange variable	description

Command Line Parameters

You can give parameters on the run command or in a config file. The default config file is `cs_init.txt` at the top level of CS (if the file exists). Or you can name where the file is on a command line parameter `config=xxx`. Config file data are command line parameters, 1 per line, like below:

```
noboot
port=20
```

Actual command line parameters have priority over config file values.

Memory options

Chatscript statically allocates its memory and so (barring unusual circumstance) will not allocate memory every during its interactions with users. These parameters can control those allocations. Done typically in a memory poor environment like a cellphone.

option	description
`buffer=50`	how many buffers to allocate for general use (80 is default)
`buffer=15x80`	allocate 15 buffers of 80k bytes each (default buffer size is 80k)

Most chat doesn't require huge output and buffers around 20k each will be more than enough. 20 buffers is often enough too (depends on recursive issues in your scripts).

If the system runs out of buffers, it will perform emergency allocations to try to get more, but in limited memory environments (like phones) it might fail. You are not allowed to allocate less than a 20K buffer size.

option	description
`dict=n`	limit dictionary to this size entries
`text=n`	limit string space to this many bytes
`fact=n`	limit fact pool to this number of facts
`hash=n`	use this hash size for finding dictionary words (bigger = faster access)
`cache=1x50`	allocate a 50K buffer for handling 1 user file at a time. A server might want to cache multiple users at a time.

A default version of ChatScript will allocate much more than it needs, because it doesn't know what you might need.

If you want to use the least amount of memory (multiple servers on a machine or running on a mobile device), you should look at the USED line on startup and add small amounts to the entries (unless your application does unusual things with facts).

If you want to know how much, try doing `:show stats` and then `:source REGRESS/bigregress.txt`. This will run your bot against a wide range of input and the stats at the end will include the maximum values needed during a volley. To be paranoid, add beyond those valuse. Take your max dict value and double it. Same with max fact. Add 10000 to max text.

Just for reference, for our most advanced bot, the actual max values used were: max dict: 346 max fact: 689 max text: 38052.

And the maximum rules executed to find an answer to an input sentence was 8426 (not that you control or care). Typical rules executed for an input sentence was 500-2000 rules. For example, add 1000 to the dict and fact used amounts and 10 (kb) to the string space to have enough normal working room.

Output options

`output=nnn` limits output line length for a bot to that amount (forcing crnl as needed). 0 is unlimited.

`outputsize=80000` is the maximum output that can be shipped by a volley from the bot without getting truncated. Actually the value is somewhat less, because routines generating partial data for later incorporation into the output also use the buffer and need some usually small amount of clearance. You can find out how close you have approached the max in a session by typing `:memstats`. If you need to ship a lot of data around, you can raise this into the megabyte range and expect CS will continue to function. 80K is the default.

For normal operation, when you change `outputsize` you should also change `logsize` to be at least as much, so that the system can do complete logs. You are welcome to set log size lots smaller if you don't care about the log.

File options

option	description
`livedata=xxx`	name relative or absolute path to your own private LIVEDATA folder. Do not add trailing / on this pathRecommended is you use `RAWDATA/yourbotfolder/LIVEDATA` to keep all your data in one place. You can have your own live data, yet use ChatScripts default `LIVEDATA/SYSTEM` and `LIVEDATA/ENGLISH` by providing paths to the `system=` and `english=` parameters as well as the `livedata=` parameter
`users=xxx`	name relative or absolute path to where you want the USERS folder to be. Do not add trailing `/`
`logs=xxx`	name relative or absolute path to where you want the LOGS folder to be. Do not add trailing `/`
`userlog`	Store a user-bot log in USERS directory (default)
`nouserlog`	Don't store a user-bot log

Execution options

option	description
`source=xxx`	Analogous to the `:source` command. The file is executed
`login=xxx`	The same as you would name when asked for a login, this avoids having to ask for it. Can be `login=george` or `login=george:harry` or whatever
`build0=filename`	Rebuild on the filename as level0 and exits with 0 on success or 4 on failure
`build1=filename`	Rebuild on the filename as level1 and exits with 0 on success or 4 on failure.Eg. ChatScript `build0=files0.txt` will rebuild the usual level 0
`debug=:xxx`	xxx runs the given debug command and then exits. Useful for `:trim`, for example or more specific `:build` commands
`param=xxx`	data to be passed to your private code
`bootcmd=xxx`	runs this command string before CSBOOT is run; use it to trace the boot process
`trace`	turn on all tracing.
`redo`	see documentation for :redo in ChatScript Debugging Manual manual
`noboot`	Do not run any boot script on engine startup

Bot variables

You can create predefined bot variables by simply naming permanent variables on the command line, using V to replace $ (since Linux shell scripts don't like $). Eg.

```
ChatScript Vmyvar=fatcat

ChatScript Vmyvar="tony is here"

ChatScript "Vmyvar=tony is here"
```

Quoted strings will be stored without the quotes. Bot variables are always reset to their original value at each volley, even if you overwrite them during a volley. This can be used to provide server-host specific values into a script.

No such bot-specific - nosuchbotrestart=true

If the system does not recognize the bot name requested, it can automatically restart a server (on the presumption that something got damaged). If you don't expect no such bot to happen, you can enable this restart using `nosuchbotrestart=true`. Default is false.

Time options

`Timer=15000` if a volley lasts more than 15 seconds, abort it and return a timeout message.

`Timer=18000x10` same as above, but more roughly, higher number after the x reduces how frequently it samples time, reducing the cost of sampling

:TranslateConcept Google API Key

`apikey=xxxxxx` is how you provide a google translate api key to :translate concept.

Security

Typically security parameters only are used in a server configuration.

`sandbox`

If the engine is not allowed to alter the server machine other than through the standard ChatScript directories, you can start it with the parameter `sandbox` which disables Export and System calls.

`nodebug`

Users may not issue debug commands (regardless of authorizations). Scripts can still do so.

```
authorize="" bunch of authorizations ""
```

The contents of the string are just like the contents of the authorizations file for the server. Each entry separated from the other by a space. This list is checked first. If it fails to authorize AND there is a file, then the file will be checked also. Otherwise authorization is denied.

`encrypt=xxxxx`
`decrypt=xxxxx`

These name URLs that accept JSON data to encrypt and decrypt user data. User data is of two forms, topic data and LTM data. LTM data is intended to be more personalized for a user, so if `encrypt` is set, LTM will be encrypted. User topic data is often just execution state of the user and potentially big, so by default it is not encrypted. You can request encryption using `userencrypt` as a command line parameter to encrypt the topic file and `ltmdecrypt` to encrypt the ltm file.

The JSON data sent to the URL given by the parameters looks like this:

```
{"datavalues": {"x": "..."}}
```

where ... is the text to encrypt or decrypt. Data from CS will be filled into the ... and are JSON compatible.

Server Parameters

Either Mac/LINUX or Windows versions accept the following command line args:

`port=xxx`

This tells the system to be a server and to use the given numeric port. You must do this to tell Windows to run as a server. The standard port is 1024 but you can use any port.

`local`

The opposite of the port command, this says run the program as a stand-alone system, not as a server.

`interface=127.0.0.1`

By default the value is `0.0.0.0` and the system directly uses a port that may be open to the internet. You can set the interface to a different value and it will set the local port of the TCP connection to what you designate.

User Data

Scripts can direct the system to store individualized data for a user in the user's topic file in USERS. It can store user variables (`$xxx`) or facts. Since variables hold only a single piece of information a script already controls how many of those there are. But facts can be arbitrarily created by a script and there is no natural limit. As these all take up room in the user's file, affecting how long it takes to process a volley (due to the time it takes to load and write back a topic file), you may want to limit how many facts each user can have written. This is unrelated to universal facts the system has at its permanent disposal as part of the base system.

`userfacts=n` limits a user file to saving only the n most recently created facts of a user (this does not include facts stored in fact sets). Overridden if the user has `$cs_userfactlimit` set to some value

User Caching

Each user is tracked via their topic file in USERS. The system must load it and write it back for each volley and in some cases will become I/O bound as a result (particularly if the filesystem is not local).

You can direct the system to keep a cache in memory of recent users, to reduce the I/O volume. It will still write out data periodically, but not every volley. Of course if you do this and the server crashes, writebacks may not have happened and some system rememberance of user interaction will be lost.

Of course if the system crashes, user's may not think it unusually that the chatbot forgot some of what happened. By default, the system automatically writes to disk every volley, If you use a different value, a user file will never be more out of date than that.

```
cache=20
cache=20x1
```

This specifies how many users can be cached in memory and how big the cache block in kb should be for a user. The default block size is `50` (50,000 bytes). User files typically are under 20,000 bytes.

If a file is too big for the block, it will just have to write directly to and from the filesystem. The default cache count is 1, telling how many users to cache at once, but you can explicitly set how many users get cached with the number after the "x". If the second number is 0, then no caching is done and users have no data saved. They remember nothing from volley to volley.

Do not use caching with fork. The forks will be hiding user data from each other.

```
save=n
```

This specifies how many volleys should elapse before a cached user is saved to disk. Default is 1. A value of 0 not only causes a user's data to be written out every volley, but also causes the user record to be dropped from the cache, so it is read back in every time it is needed (handy when running multi-core copies of chatscript off the same port).

Note, if you change the default to a number higher than 1, you should always use `:quit` to end a server. Merely killing the process may result in loss of the most recent user activity.

Logging or Not

In stand-alone mode the system logs what a user says with a bot in the USERS folder. It can also do this in server mode. It can also log what the server itself does. But logging slows down the system. Particularly if you have an intervening server running and it is logging things, you may have no use whatsoever for ChatScript's logging.

`Userlog`

Store a user-bot log in USERS directory. Stand-alone default if unspecified.

`Nouserlog`

Don't store a user-bot log. Server default if unspecified.

`Serverlog`

Write a server log. Server default if unspecified. The server log will be put into the LOGS directory under serverlogxxx.txt where xxx is the port.

`Noserverprelog`

Normally CS writes of a copy of input before server begins work on it to server log. Helps see what crashed the server (since if it crashes you get no log entry). This turns it off to improve performance.

`Serverctrlz`

Have server terminate its output with 0x00 0xfe 0xff as a verification the client received the entire message, since without sending to server, client cannot be positive the connection wasn't broken somewhere and await more input forever.

`Noserverlog`

Don't write a server log.

`Fork=n`

If using LINUX EVSERVER, you can request extra copies of ChatScript (to run on each core for example). n specifies how many additional copies of ChatScript to launch.

`Serverretry`

Allows `:retry` to work from a server - don't use this except for testing a single-person on a server as it slows down the server.

No such bot-specific - nosuchbotrestart=true

If the system does not recognize the bot name requested, it can automatically restart a server (on the presumption that something got damaged). If you don't expect no such bot to happen, you can enable this restart using `nosuchbotrestart=true`. Default is false.

Testing a server

There are various configurations for having an instance be a client to test a server.

`client=xxxx:yyyy`

This says be a client to test a remote server at IP xxxx and port yyyy. You will be able to "login" to this client and then send and receive messages with a server.

`client=localhost:yyyy`

This says be a client to test a local server on port yyyy. Similar to above.

`Load=1`

This creates a localhost client that constantly sends messages to a server. Works its way through `REGRESS/bigregress.txt` as its input (over 100K messages). Can assign different numbers to create different loading clients (e.g., load=10 creates 10 clients).

`Dual`

Yet another client. But this one feeds the output of the server back as input for the next round. There are also command line parameters for controlling memory usage which are not specific to being a server.