ChatScript Advanced User's Manual

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

Revision 1/05/2016 cs6.01

Table of Contents

ADVANCED CONCEPTS

Concepts can have part of speech information attached to them (using dictionarysystem.h values). Eg.

concept: ~mynouns NOUN NOUN_SINGULAR (boxdead foxtrot)

concept: ~myadjectives ADJECTIVE ADJECTIVE_BASIC (moony dizcious)

Since the script compile issues warning messages on words it doesn't recognize, in case you misspelled them, you can also add IGNORESPELLING as a flag on the concept:

concept: ~unknownwords IGNORESPELLING (asl daghh)

and you can combine pos declarations and ignorespelling. This is applied recursively to any concepts that are members of this concept.

That may be a bit excessive. Rather than assigning parts of speech you can recursively limit a concept's words to a part of speech using ONLY_NOUNS, ONLY_VERBS, ONLY_ADJECTIVES, or ONLY_ADVERBS.

Concept: ~verbs ONLY_VERBS (sit sleep)

This will not react to noun meanings of sleep. The current ontology files for verbs, adverbs, and adjectives all have the appropriate ONLY marked on them. When you don't want a member concept marked as a consequence, you can use ONLY_NONE to block propogate. Thus:

concept: ~verbs ONLY_VERBS (~active_verbs sit)

concept: ~active_verbs ONLY_NONE (sleep)

will prevent sleep from being required to be a verb form. Note that verb forms do not include verbs used as nouns (ie gerunds).

Normally if you declare a concept a second time, the system considers that an error. If you add the marker MORE to its definition, it will allow you to augment an existing list.

concept: ~city MORE (Tokyo)

Normally concepts (and topics) discard repeated keywords. For concepts, you can force it to allow repeats using DUPLICATE

concept: ~mapword DUPLICATE (year month day year month day) – the concept has 6 members

ADVANCED TOPICS

There are several things to know about advanced topics.

Topic Execution

When a topic is executing rules, it does not stop just because a rule matches. It will keep executing rules until some rule generates ouput for the user or something issues an appropriate ^end or ^fail call. So you can do things like this:

u: (I love) $userloves = true

u: (dog) $animal = dog

u: (love) Glad to hear it

u: (dog) I hate dogs

and given “I love dogs”, the system will set $userloves and $animal and output *glad to hear it.*

Topic Control Flags

The first are topic flags, that control a topic's overall behavior. These are placed between the topic name and the (keywords list). You may have multiple flags. E.g.
topic: ~rust keep random [rust iron oxide]
The flags and their meanings are:

Random - search rules randomly instead of linearly
NoRandom – (default) search rules linearly
Keep – do not erase responders ever. Gambits (and rejoinders) are not affected by this.
Erase – (default) erase responders that successfully generate output. (Gambits automatically erase unless you suppress them specifically.
NoStay – do not consider this a topic to remain in, leave it (except for rejoinders).
Stay – (default) make this a pending topic when it generates output
Repeat - allow rules to generate output which has been output recently
NoRepeat – (default) do not generate output if it matches output made recently
Priority- raise the priority of this topic when matching keywords
Normal- (default) give this topic normal priority when matching keywords
Deprioritize – lower the priority of this topic when matching keywords
System – this is a system topic. It is automatically NoStay, Keep. Keep automatically applies to gambits as well. The system never looks to these topics for gambits. System topics can never be considered pending (defined shortly). They can not have themselves or their rules be enabled or disabled. Their status/data is never saved to user files.
User - (default) this is a normal topic.
NoBlocking - :verify should not perform any blocking tests on this topic
NoPatterns - :verify should not perform any pattern tests on this topic
NoSamples - :verify should not perform any sample tests on this topic
NoKeys - :verify should not perform any keyword tests on this topic
More – Normally if you try to redeclare a concept, you get an error. MORE tells CS you intend to extend the topic and allows additional keywords for it.
Bot=name – if this is given, only named bots are allowed to use this topic. You can name multiple bots separated by commas with no extra spaces. E.g.,
topic: ~mytopic bot=harry,,roman [mykeyword]

To support multiple bots, you may create multiple copies of a topic name, which vary in their bot restrictions and content. The set of keywords given for a topic is the union of the keywords of all copies of that topic name. You should not name a topic ending in a period and a number (that is used internally to represent multiple topics of the same name).

bot: name - There is also a top level command you can put in a file to label all topics thereafter with a uniform bot restriction. Bot: name will do this, unless the topic has an explicit *bot=*
You can, for example, put at the topic of simpletopic.top the command

bot: harry,georgia

and get all topics in that file restricted to the two bots named. There should be no spaces after the comma. Of course there are only two bots in the first place, so this doesn't actually accomplish anything useful.

Share – Normally, if you have multiple bots, they all talk independently to the user. What one learns or says is not available to the other bots. It is possible to create a collection of bots that all can hear what has been said and share information. **Share** on a topic means that all bots will have common access/modification of a topic's state. So if one bot uses up a gambit, the other bots will not try to use that gambit themselves. All facts created will be visible to all bots. And if you create a permanent user variable with the starting name $share_, then all bots can see and modify it. So $share_name becomes a common variable. When sharing is in effect, the state with the user (what he said, what bot said, what turn of the volley this is, where the rejoinder mark is) is all common among the bots- they are advancing a joint conversation.

Rules that erase and repeat

Normally a rule that successfully generates output directly erases itself so it won't run again. Gambits do this and responders do this.

Gambits will erase themselves even if they don't generate output. They are intended to tell a story or progress some action, and so do their thing and then disappear automatically.

Rejoinders don't erase individually, they disappear when the rule they are controlled by disappears. A rule that is marked keep will not erase itself. Nor will responders in a topic marked keep (but gambits still will).

Responders that generate output erase themselves. Responders that cause others to generate output will not normally erase themselves (unless...):

u: (*) respond(~reactor)

If the above rule causes output to be generated, this rule won't erase itself, the rule invoked from the ~reactor topic that actually generated the output will erase itself. But, if the rule generating the output is marked keep, then since someone has to pay the price for output, it will be this calling rule instead.

Repeat does not stop a rule from firing, it merely suppresses its output. So the rule fires, does any other effects it might have, but does not generate output. For a responder, if it doesn't generate output, then it won't erase itself. For a gambit, it will because gambits erase themselves regardless of whether they generate output or not.

Keywords vs Control Script

A topic can be invoked as a result of its keywords or by a direct call from the control script or some other topic. If you intend to call it from script, then there is almost never any reason to give it keywords as well, because that may result in it being called twice,

which is wasteful, or out of order, if there was a reason for the point you called it from script.

Pending Topics

The second thing to know about topics is what makes a topic pending. Control flow passes through various topics, some of which become pending, meaning one wants to continue in those topics when talking to the user. Topics that can never be pending are: system topics, blocked topics (you can block a topic so it won't execute), and nostay topics.

What makes a remaining topic pending is one of two things. Either the system is currently executing rules in the topic or the system previously generated a user response from the topic. When the system leaves a topic that didn't say anything to the user, it is no longer pending. But once a topic has said something, the system expects to continue in that topic or resume that topic.

The system has an ordered list of pending topics. The order is: 1st- being within that topic executing rules now, 2nd- the most recently added topic (or revived topic) is the most pending.. You can get the name of the current most pending topic (*%topic*), add pending topics yourself (*^addtopic*()), and remove a topic off the list (*^poptopic*()).

Random Gambit

The third thing about topics is that they introduce another type, the random gambit, *r:*.

The topic gambit *t:* executes in sequence forming in effect one big story for the duration of the topic. You can force them to be dished randomly by setting the *random* flag on the topic, but that will also randomize the responders. And sometimes what you want is semirandomness in gambits. That is, a topic treated as a collection of subtopics for gambit purposes.

This is *r:* The engine selects an *r:* gambit randomly, but any *t:* topic gambits that follow it up until the next random gambit are considered "attached" to it. They will be executed in sequence until they are used up, after which the next random gambit is selected.

```
Topic: ~beach [beach sand ocean sand_castle]
# subtopic about swimming
r: Do you like the ocean?
t: I like swimming in the ocean.
t: I often go to the beach to swim.
# subtopic about sand castles.
r: Have you made sand castles?
        a: (~yes) Maybe sometime you can make some that I can go see.
        a: (~no) I admire those who make luxury sand castles.
t: I've seen pictures of some really grand sand castles.
```

This topic has a subtopic on swimming and one on sand castles. It will select the subtopic

randomly, then over time exhaust it before moving onto the other subtopic.

Note any t: gambits occurring before the first r: gambit, will get executed linearly until the r: gambits can fire.

Overview of the control script

Normally you start using the system with the pre-given control script. But it's just a topic and you can modify it or write your own.

The typical flow of control is for the control script to try to invoke a pending rejoinder. This allows the system to directly test rules related to its last output, rules that anticipate how the user will respond. Unlike responders and gambits, the engine will keep trying rejoinders below a rule until the pattern of one matches and the output doesn't fail. Not failing does not require that it generate user output. Merely that it doesn't return a fail code. Whereas responders and gambits are tried until user output is generated (or you run out of them in a topic).

If no output is generated from rejoinders, the system would test responders. First in the current topic, to see if the current topic can be continued directly. If that fails to generate output, the system would check other topics whose keywords match the input to see if they have responders that match. If that fails, the system would call topics explicitly named which do not involve keywords. These are generic topics you might have set up.

If finding a responder fails, the system would try to issue a gambit. First, from a topic with matching keywords. If that fails, the system would try to issue a gambit from the current topic. If that fails, the system would generate a random gambit.

Once you find an output, the work of the system is nominally done. It records what rule generated the output, so it can see rejoinders attached to it on next input. And it records the current topic, so that will be biased for responding to the next input. And then the system is done. The next input starts the process of trying to find appropriate rules anew.

There are actually three control scripts (or one invoked multiple ways). The first is the preprocess, called before any user sentences are analyzed. The main script is invoked for each input sentence. The postprocess is invoked after all user input is complete. It allows you to examine what was generated (but not to generate new output).

ADVANCED PATTERNS

Keyword Phrases

You cannot make a concept out with a member whose string includes starting or trailing blanks, like " X ". Such a word could never match as a pattern, since spaces are skipped over. But you can make it respond to idiomatic phrases and multiple words. Just put them in quotes. E.g., concept: ~remove ("take away" remove)

Normally in patterns you can write

?: (do you take away cheese)

and the system will match sentences with those words in order. In WordNet, some words are actually composite words like : TV_show. When you do :prepare on *what is your favorite TV show* you will discover that the engine has merged TV_show into one composite word. The system has no trouble matching inputs where the words are split apart

?: (what is your favorite TV show)

But if you tried a word memorize like

*?: (what is your favorite *1 *1)*

that would fail because the first *1 memorizes *TV_show* and there is therefore no second word to memorize. Likewise when you write *concept: ~viewing ("TV show")* the system can match that concept readily also. In fact, whenever you write the quoted keyword phrase, if all its words are canonical, you can match canonical and noncanonincal forms. "TV show" matchs *TV shows* as well as *TV show.*

Dictionary Keyword sets

In ChatScript, WordNet ontologies are invoked by naming the word, a ~, and the index of the meaning you want.

concept: ~buildings [shelter~1 living_accomodations~1 building~3]

The concept *~buildings* represents 760 general and specific building words found in the WordNet dictionary – any word which is a child of: definition 1 of shelter, definition 1 of accommodations, or definition 3 of building in WordNet's ontology. How would you be able to figure out creating this? This is described under *:up* in **Word Commands** later. *Building~3* and *building~3n* are equivalent. The first is what you might say to refer to the 3rd meaning of building. Internally *building~3n* denotes the 3rd meaning and its a noun meaning. You may see that in printouts from Chatscript. If you write 3n yourself, the system will strip off the n marker as superfluous.

Similarly you can invoke parts of speech classes on words. By default you get all of them. If you write: *concept: ~beings [snake mother]*
then a sentence like *I like mothering my baby* would trigger this concept, as would *He snaked his way through the grass.* But the engine has a dictionary and a part-of-speech tagger, so it often knows what part of speech a word in the sentence is. You can use that to help prevent false matches to concepts by adding *~n ~v ~a* or *~b* (adverb) after a word.

concept: ~beings [snake~n mother~n]

If the system isn't sure something is only a noun, it would let the verb match still. Thus a user single-word reply of *snakes* would be considered both noun and verb.

The notation run~46 exists to represent a meaning. There is mild inherent danger that I might kill off some word meaning that is problematic (eg if run~23 turned out mark the ~curses set and I didn't want the resulting confusion), said kill off might strand your meaning by renumbering into either non-existence (in which case the script compiler will warn you) or into a different pos set (because your meaning was on the boundary of meanings of a different pos type). Use of the specific meaning is handy in defining concepts when the meaning is a noun, because Wordnet has a good noun ontology. Use of the specific meaning of other parts of speech is more questionable, as Wordnet does not have much ontology for them.

The broader scope meaning restriction by part-of-speech (eg run~v) has much more utility. It has its risks in that it depends on the parser getting it right (as you have seen), which over time will get better and better. In MOST cases, you are better off with the full fledged unadorned word, which is parse-independent. This is particularly true when you are pattern matching adjacent words and so context is firm. < run ~app is a pretty clean context which does not need pos-certification. The topic on ~drugs would want in its keyword list clean~a to allow "I've been clean for months" to target it, but not "I clean my house".

System Functions

You can call any predefined system function. It will fail the pattern if it returns any fail or end code. It will pass otherwise. The most likely functions you would call would be:

^*query* – to see if some fact data could be found.

Many functions make no sense to call, because they are only useful on output and their behavior on the pattern side is unpredictable.

Macros

Just as you can use sets to "share" data across rules, you can also write macros to share code. A patternmacro is a top-level declaration that declares a name, arguments that can be passed, and a set of script to be executed "as though the script code were in place of the original call". Macro names can be ordinary names or have a ^ in front of them. The arguments must always begin with ^. The definition ends with the start of a new top-level declaration or end of file. E.g.

```
patternmacro: ^ISHAIRCOLOR(^who)
![not never]
[
( << be ^who [blonde brunette redhead blond ] >> )
(<< what ^who hair color >> )
]

?: (^ISHAIRCOLOR(I)) How would I know your hair color?
```

The above patternmacro takes one argument (who we are talking about). After checking

that the sentence is not in the negative, it uses a choice to consider alternative ways of asking what the hair color is. The first way matches *are you a redhead*. The second way matches *what is my hair color*. The call passes in the value *I* (which will also match *my mine* etc in the canonical form). Every place in the macro code where ^who exists, the actual value passed through will be used.

You cannot omit the ^ prefix in the call. The system has no way to distinguish it otherwise.

Whereas most programming language separate their arguments with commas because they are reserved tokens in their language, in ChatScript a comma is a normal word. So you separate arguments to functions just with spaces.

?: (^FiveArgFunction(1 3 my , word))

When a patternmacro takes a single argument and you want to pass in several, you can wrap them in parens to make them a single argument. Or sometimes brackets. E.g.,

*?: (^DoYouDoThis((play * baseball))) Yes I do*
?: (^DoYouDoThis([swim surf "scuba dive"]) Yes I do

If you call a patternmacro with a string argument, like "scuba dive" above, the system will convert that to its internal single-token format just as it would have had it been part of a normal pattern. Quoted strings to output macros are treated differently and left in string form when passed.

You can declare a patternmacro to accept a variable number of arguments. You define the macro with the maximum and then put "variable" before the argument list. All missing arguments will be set to null on the call.

patternmacro: ^myfn variable (^arg1 ^arg2 ^arg3 ^arg4)

You can also declare something dualmacro: which means it can be used in both pattern and output contexts.

Patternmacro cannot be passed a factset name. These are not legal calls:

^mymacro(@0)
^mymacro(@0subject)

**Literal Next **

If you need to test a character that is normally reserved, like (or [, you can put a backslash in front of it.

*s: (\(* \)) Why are you saying that aside?*

This also works with entire tokens like:

u: (\test=fort)

Normally the above without \ would be considered a comparison. But the \ at the start of it says treat = as just an ordinary part of the token. You can even put _ in front of it:

u: (_\test=fort)

Note that \ does not block a word with an * in it from performing wildcard spelling.

Question and exclamation ? '!

Normally you already know that an input was a question because you used the rule type ?: . But rejoinders do not have rule types, so if you want to know if something was a question or not, you need to use the ? keyword. It doesn't change the match position

t: Do you like germs?

a: (?) Why are you asking a question instead of answering me?

a: (!?) I appreciate your statement.

If you want to know if an exclamation ended his sentence, just backslash a ! so it won't be treated as a not request. This doesn't change the match position.

s: (I like \) Why so much excitement

More comparison tests & and ?

You can use the logical *and* bit-relation to test numbers. Any non-zero value passes.

s: (_~number _0&1) Your number is odd.

? can be used in two ways. As a comparison operator, it allows you to see if the item on the left side is a member of a set on the right. E.g.

u: (_~propername?~bands)

As a standalone, it allows you to ask if a wildcard or variable is in the sentence. E.g.

u: (_1?)

u: ($bot?)

Note that when _1 is a normal word, that is simple for CS to handle. If _1 is a phrase, then generally CS cannot match it. This is because for phrases, CS needs to know in advance that a phrase can be matched. If you put "take a seat" as a keyword in a concept or topic or pattern, that phrase is stored in the dictionary and marked as a pattern phrase, meaning if the phrase is ever seen in a sentence, it should be noticed and marked so it can be matched in a pattern. But if it is merely in a variable, then the dictionary is unaware of the phrase and so _1? will not work for it.

Comparison with C++ #define in dictionarysystem.h

You can name a constant from that file as the right hand side of a comparison test by prefixing its name with #. E.g.,

s: (_~number _0=#NOUN)

Such constants can be done anywhere, not just in a pattern.

Current Topic ~

Whenever you successfully execute a rule in a topic, that topic becomes a pending topic (if the topic is not declared system or nostay). When you execute a rule, while the rule is obviously in a topic being tested, it is not necessarily a topic that was in use recently.

You can ask if the system is currently in a topic (meaning was there last volley) via ~. if the topic is currently on the pending list, then the system will match the ~. E.g.,

u: (chocolate ~) I love chocolate ice cream.

The above does not match any use of chocolate, only one where we are already in this topic (like topic: ~ice_cream) or we were recently in this topic and are reconsidering it now.

A useful idiom is [~topicname ~]. This allows you to match if EITHER the user gave keywords of the topic OR you are already in the topic. So:

u: (<<chocolate [~ice_cream ~] >>)

would match if you only said "chocolate" while inside the topic, or if you said "chocolate ice cream" while outside the topic.

Prefix Wildcard Spelling and Postfix Wildcard Spelling

Some words you just know people will get wrong, like Schrodinger's cat or Sagittarius. You can request a partial match by using an * to represent any number of letters (includig 0). For example

u: (Sag)* This matches Sagittarius in a variety of misspellings.

*u: (*tor)* this matches "reactor".

The * can occur in one or more positions and means 0 or more letters match. A period can be used to match a single letter (but may not start a wildcardprefix). E.g.,

u: (p.t)*

can match *pituitary* or merely *pit* or *pat*. You cannot use a wildcard on the first letter of the pattern.

u: (.p)* is not legal because it may not start with a period.

Indirect pattern elements

Most patterns are easy to understand because what words they look at is usually staring you in the face. With indirection, you can pass patterndata from other topics, at a cost of obscurity. Declaring a macro does this. A ^ normally means a macro call (if what follows it is arguments in parens), or a macro argument. The contents of the macro argument are used in the pattern in its place. Macro arguments only exist inside of macros. But macros don't let you write rules, only pieces of rules.

Normally you use a variable directly. $$tmp = nil clears the $$tmp variable, for instance, while *u: ( ) $$tmp goes home* will output the value into the output stream.

The functional user argument lets you pass pattern data from one topic to another.

s: (are you a _^$var)

The contents of $var are used at that point in the pattern. Maybe it is a set being named. Maybe it's a word. You can also do whole expressions, but if you do you will be at risk because you won't have the script compiler protecting you and properly formatting your data. See also Advanced Variables.

Setting Match Position - @_3+ @_3-

You can “back up” and continue matching from a prior match position using @ followed by a match variable previously set. E.g.

*u: (_~pronoun * go @_0+ often)*

This matches “I often go” but not “I go”

Just as < sets the position pointer to the start, @_0+ makes the pattern think it just matched that wildcard at that location in the sentence going forward.

s: (_is _~she) # for input: *is mother going* this sets *_0* to is and *_1* to mother

s: (@_1+ going) # this completes the match of *is mother going*

OK. Setting positional context is really obscure and probably not for you. So why does it exist? It supports shared code for pseudo parsing.

You can match either forwards or backwards. Normally matching is always done forwards. But you can set the direction of matching at the same time as you set a position. Forward matching is @_n+ and backward matching is @_n-.

Backward Wildcards

You can request n words before the current position using *-n. For example

*u: (I love * > _*-1)* capture last word of sentence

Gory details about strings

u: (I “take charge”) OK.

When you use "take charge" it can match taking charges, take charge, etc. When you use "taking charge" it can only match that, not "taking charges'. If all words in the string are canonical, it can cover all forms. If any are not, it can only literally match.

The quote notation is typically used in several situations...
1. you are matching a recognized phrase so quoting it emphasizes that
2. what you are matching contains embedded punctuation and you don't want to think about how to tokenize it e.g., "Mrs. Watson's" -- is the period part of Mrs, is the ' part of Watson, etc. Easier just to use quotes and let the system handle it.
3. You want to use a phrase inside [] or {} choices. Like [like live “really enjoy”]
4.

In actuality, when you quote something, the system generates a correctly tokenized set of words and joins them with underscores into a single word.nThere is no real difference between "go back" and go_back in a pattern.

But compared to just listing the words in sequence in your pattern, a quoted expression cannot handle optional choices of words. You can't write "go {really almost} back" where that can match *go back* or *go almost back*. So there is that limitation when using a string inside [] or {}. But, one can write a pattern for that. While [] means a choice of words and { } means a choice of optional words, ( ) means these things in sequence. So

you could write:

u: ([next before (go {almost really} back)])

and that will be a more complex pattern. One almost never has a real use for that capability, but you use () notation all the time, of course. In fact, all rules have an implied < * in front of the (). That's what allows them to find a sequence of words starting anywhere in the input. But when you nest () inside, unless you write < * yourself, you are committed to remaining in the sequence set up.

As a side note, the quoted expression is faster to match than the ( ) one. That's because the cost of matching is linear in the number of items to test. And a quoted expression (or the _ equivalent) is a single item, whereas (take charge) is 4 items. So the first rule will below will match faster than the second rule:

u: ("I love you today when")

u: (I love you today when)

But quoted expressions only work up to 5 words in the expression (one rarely has a need for more) whereas () notation takes any number. And using quotes when it isn't a common phrase is obscure and not worth doing.

Generalizing a pattern word

When you want to generalize a word, a handy thing to do is type :concepts word and see what concepts it belongs to, picking a concept that most broadly expresses your meaning. The system will show you both concepts and topics that encompass the word. Because topics are more unreliable (contain or may in the future contain words not appropriate to your generalization, topics are always shown a T~ rather than the mere ~name.

The deep view of patterns

You normally think of the pattern matcher as matching words from the sentence. It doesn't. It matches marks. A mark is an arbitrary text string that can be associated with a sentence location. The actual words (but not necessarily the actual case you use) are just marks placed on the actual locations in the sentence where the words exist. The canonical forms of those words are also such marks. As are the concept set names and topic names which have those words as keywords. And all parser determined pos-tag and parser roles of words.

It gets interesting because marks can also cover a sequential range of locations. That's how the system detects phrases as keywords, idioms from the parser like *I am a little bit crazy* (where *a little bit* is an adverb covering 3 sentence locations) and contiguous phrasal verbs.

And there are functions you can call to set or erase marks on your own (^mark and ^unmark).

Pattern matching involves looking at tokens while having a current sentence location index. Unless you are using a wildcard, your tokens must occur in contiguous order to match. As they match, the current sentence location is updated. But not necessarily

updated to the next adjacent location. It will depend on the length of the mark being matched. So your token might be ~adverb but that may match a multiple-word sequence, so the location index will be updated to the end of that index.

And you can play with the location index itself, setting it to the location of a previously matched _ variable and setting whether matching should proceed forwards or backwards through the sentence. Really, the actual capabilities of pattern matching are quite outrageous.

Pattern matching operates token by token. If you have a pattern:

u: ({blue} sky is blue)

and you input "the sky is blue", this pattern will fail, even though the initial {blue} is optional. Optional should not lead a pattern, it is used for word alignment. The first blue is found and so the system locks itself at that point. It then looks to see if the next word is sky. It's not. Pattern fails. It then unlocks itself and allows trying to match from the start of the pattern one later. So {blue} matches blue at the end of the sentence. It locks itself. The next word is not sky. Pattern fails. There is nothing left to try.

Interesting thing about match variables

Unlike user variables, which are saved with users, match variables (like _0) are global to ChatScript. They are initialized on startup and never destroyed. They are overwritten by a rule that forces a match value onto them and by things like _0 = ^burst(..) or _3 = ^first(@0all). And those may overrun the needed number of variables by 1 to indicate the end of a sequence. But this means typically _10 and above are easily available as permanent variables that can hold values across all users etc. This might be handy in a server context and is definitely handy in :document mode where user memory can be very transient. Of course remembering what _10 means as a holding variable is harder, unlike the ones bound to matches from input. So you can use

rename: _bettername _12

in a script before any uses of _bettername, which now mean _12.

ADVANCED OUTPUT

Simple output puts words into the output stream, a magical place that queues up each word you write in a rule output. What I didn't tell you before was that if the rule fails along the way, an incomplete stream is canceled and says nothing to the user.
For example,

t: I love this rule. ^fail(RULE)

Processing the above gambits successively puts the words "I", "love", "this", "rule", "." into the output stream of that rule. If somewhere along the way that rule fails (in this case by the call at the end), the stream is discarded. If the rule completes and this is a top level rule, the stream is converted into a line of output and stored in the responses list. When the system is finished processing all rules, it will display the responses list to the user, in the order they were generated (unless you used ^preprint or ^insertprint to generate responses in a different order). If the output was destined for storing on a variable or becoming the argument to a function or macro, then the output stream is

stored in the appropriate place instead.

I also didn't tell you that the system monitors what it says, and won't repeat itself (even if from a different rule) within the last 20 outputs. So if, when converting the output stream into a response to go in the responses list, the system finds it already had such a response sent to the user in some recently earlier volley, the output is also discarded and the rule "fails".

Actually, it's a bit more complicated than that. Let's imagine a stream is being built up. And then suddenly the rule calls another rule (^reuse, ^gambit, ^repond). What happens? E.g., u: (some test) I like fruit and vegetables. ^reuse(COMMON) And so do you. What happens is this- when the system detects the transfer of control (the ^reuse call), if there is output pending it is finished off and packaged for the user. The current stream is cleared, and the rule is erased (if allowed to be). Then the ^reuse() happens. Even if it fails, this rule has produced output and been erased. Assuming the reuse doesn't fail, it will have sent whatever it wants into the stream and been packaged up for the user. The rest of the message for this rule now goes into the output stream ("and so do you") and then that too is finished off and packaged for the user. The rule is erased because it has output in the stream when it ends (but it was already erased so it doesn't matter).

So, output does two things. It queues up tokens to send to the user, which may be discarded if the rule ultimately fails. And it can call out to various functions. Things those functions may do are permanent, not undone if the rule later fails.

Formatted double quotes (Format String)

Because you can intermix program script with ordinary output words, ChatScript normally autoformats output code. But programming languages allow you to control your output with format strings and ChatScript is no exception. In the case of ChatScript, the functional string ^*"xxx"* string is a format string. The system will remove the ^ and the quotes and put it out exactly as you have it, *except*, it will substitute variables (which you learn about shortly) with their values and it will accept [ ] [] choice blocks. And will allow function calls. You can't do assignment statements or loops or if statements.

t: ^"I like you."

puts out *I like you*.

When you want special characters in the format string like [ ] and ( ), you need to backslash them, which the format string removes when it executes. For example:

u: () $tmp = [hi ^"there [ ] john"]

the brackets inside the format string will confuse the choices code of output unless you protect it using \[and \] .

Format strings evaluate themselves as soon as they can. If you write:

u: () $tmp = ^" This is $var output"

then $tmp will be set to the result of evaluating the format string. Similarly, if you write:

u: () $tmp = ^myfunc(^" This is $var output")

then the format string is evaluated before being sent to ^myfunc.

You can continue a format string across multiple source lines. It will always have a single space representing the line change, regardless of how many spaces were before or after the line break. E.g

^*"this is*

my life" → ^"this is my life"

regardless of whether there were no spaces after is or 100 spaces after is. You may not have comments at the ends of such lines (they would be absorbed into the string).

Functional Strings

Whenever format strings are placed in tables, they become a slightly different flavor, called functional strings. They are like regular output- they are literally output script. Formatting is automatic and you get to make them do any kind of executable thing, as though you were staring at actual output script. So you lose the ability to control spacing, but gain full other output execution abilities.

They have to be different because there is no user context when compiling a table. As a consequence, if you have table code that looks like this:

^createfact(x y ^" This is $var output")

the functional string does NOT evaluate itself before going to createfact. It gets stored as its original self.

We will now learn functions that can be called that might fail or have interesting other effects. And some control constructs.

Loop Construct – loop or **^loop**

Loop allows you to repeat script. It takes an optional argument within parens, which is how many times to loop. It executes the code within { } until the loop count expires or until a FAIL or END code of some kind is issued. End(loop) signals merely the end of the loop, not really the rule, and will not cancel any pending output in the output stream. Fail(LOOP) will terminate both loop and rule enclosing. All other return codes have their usual effect. Deprecated is fail(rule) and end(rule) which merely terminated the loop.

t: Hello. ^loop (5) { me }

t: ^loop () { This is forever, not. end(LOOP)}

The first gambit prints *Hello. me me me me me.* The second loop would print forever, but actually prints out *This is forever, not.* because after starting output, the loop is terminated. Loop also has a built in limit of 1000 so it will never run forever. You can override this if you define $cs_looplimit to have some value you prefer.

^loop(n)

Loop can be given a count. This can be either a number, or you can use a factset id to loop through each item of the factset via

^loop (@0)

If Construct - if or **^if**

The if allows you to conditionally execute blocks of script. The full syntax is:

If (test1) { script1 } else if (test2) { script2 } ... else { script3 }

You can omit the *else if* section, having just *if* and *else,* and you can omit the *else* section, having just *if* or *if* and *else if*. You may have any number of *else if* sections.
The test condition can be:

1. A variable – if it is defined, the test passes
2. ! variable – if it is not defined, the test passes (same as relation variable == null)
3. A function call – if it doesn't fail or return the values 0 or null or nil, it passes
4. A relation – one of == != < <= > >= ? !?

For the purposes of numeric comparison (< <= > >=) a null value compared against a number will be considers as 0.

You may have a series of test conditions separated by AND and OR. The failure of the test condition can be any *end* or *fail* code. It does not affect outside the condition; it merely controls which branch of the *if* gets taken.

if ($var) { } # if $var has a value
if ($var == 5 and foo(3)) {} # if $var is 5 and foo(3) doesn't fail

Comparison tests between two text strings is case insensitive.

A word of warning on the ? (in set) relation test. It only works for actual concepts that have enumerated values. A number of sets marked by the engine for patterns do not consist of enumerated members. All of the pos-tagging and parse-related concepts are like this, so you cannot use ~number, ~noun, ~verb, etc here.

Quoting

Normally output evaluates things it sees. This includes $user variables, which print out their value. But if you put quote in front of it, it prints its own name. '$name will print $name. The exception to this rule is that internal functions that process their own arguments uniquely can do what they want, and the **query** function defines '$name to mean use the contents of $name, just don't expand it if it is a concept or topic name as value.

Similarly, a function variable like ^name will pretend it was its content originally. This means if the value was $var then, had $var been there in the output originally, it would have printed out its content. So normally ^name will print out the contents of its content. Again, you can suppress with using '^name to force it to only print its content directly.

Outputting underscores

Normal English sentences do not contain underscores. Wordnet uses underscores in composite words involving spaces. ChatScript, therefore has a special use for underscores

internally and if you put underscores in your output text, when they are shipped to the user they are converted to spaces. This doesn't apply to internal uses like storing on variables. So normally you cannot output an underscore to a user. But a web address might legitimately contain underscores. So, if you put two underscores in a row, ChatScript will output a single underscore to the user.

Response Controlinput

Having said that CS automatically changes underscores to spaces, you can alter this and other default response output processing. The variable "$cs_response" can be set to some combination of values to alter behavior. The default value is

$cs_response = #RESPONSE_UPPERSTART +
#RESPONSE_REMOVESPACEBEFORECOMMA +
#RESPONSE_ALTERUNDERSCORES

which controls automatically up-casing the first letter of output, removing spaces before commas, and converting underscores to spaces (and also removing ~ from concept names). Equivalently $cs_response = #ALL_RESPONSES, which if you want all is what you should use in case new ones are added in the system later.

Output Macros

Just as you can write your own common routines for handling pattern code with *patternmacro:*, you can do the same for output code. *Outputmacro: name (^arg1 ^arg2 ...)* and then your code. Only now you use output script instead of pattern stuff. Again, when calling the macro, arguments are separated with spaces and not commas. Whereas most programming language separate their arguments with commas because they are reserved tokens in their language, in ChatScript a comma is a normal word. So you separate arguments to functions just with spaces.

?: (hi) ^FiveArgFunction(1 3 my , word)

Outputmacros can return a value, just like a normal function. You just dump the text as you would a message to the user.

outputmacro: ^mymac()
tested here
TOPIC: ~patterns keep repeat []
#! what time is it?
u: (<< what time >>) $test = ^mymac() join(ok $test)

will print "oktested here".

dualmacro: name(...) is exactly like outputmacro:, but the function can be called on then side or on the output side. Typically this makes most sense for a function that performs a fixed ^query which you can see if it fails in pattern side or as a test on the output side or inside an if condition.

Output macros can be passed system function names and output macro names, allowing you to indirectly call things. E.g.,

outputmacro: ^indirect(^fn ^value)

$$tmp = ^fn(^value)

Outputmacros cannot be passed a factset name. These are not legal calls:

^mymacro(@0)
^mymacro(@0subject)

The above will evaluate ^fn, and if it finds that it is a function name, will use that in a call. The only tricky part is creating the name of the function to call in the first place. If you just write a function name with ^ in output, the compiler will complain about your call. So you have to synthesize the name somehow. Here are two ways:

outputmacro: ^mycall()
$$tmpfn = ^join(^"\^" func)
^indirect($$tmpfn 34)
^indirect(^"\^func" 34)

You can also store function names on user and match variables and then call them. E.g.,

$$tmp = ^"\^func"
$$tmp(34)

You can declare an outputmacro to accept a variable number of arguments. You define the macro with the maximum and then put "variable" before the argument list. All missing arguments will be set to null on the call.

outputmacro: ^myfn variable (^arg1 ^arg2 ^arg3 ^arg4)

Output Macros vs ^reuse()

An outputmacro is a block of code, treated as a kind of function. But another way to make a block of code is create a rule and ^reuse it. E.g.

s: MYCODE (?) here is a block of code that goes on and on

Code you write in an output macro could be code you write on the output side of a ^reused rule. Notice that this rule can never trigger on its own (an input sentence cannot be a statement and a question simultaneously). So what are the distinctions between the two?

The distinction is not in tracing. You can trace a single rule or an outputmacro equally. And both return to their caller when normally complete.

An advantage for outputmacros is that you can pass them arguments, making them more easily tailorable. To do the same with a rule, you have to store values on globals, which is more inconvenient, harder to understand, and subject to the risk that you accidentally reuse the same variable in something the rule calls. Similarly outputmacros can return a value, while you have to use globals to return a value from a rule.

An advantage for rules is that they can have rejoinders. So if the rule generates output, it may also have rejoinders to react to it. Another advantage for rules is that you can terminate their execution early ^end(RULE) without impacting the calling rule. There is no such ability in an outputmacro, so you'd have to organize if statements to manage early termination effects.

System Functions

There are many system functions to perform specific tasks. These are enumerated in the ChatScript System Function manual and the ChatScript Facts manual.

Randomized Output Revisited []

Remember this construct:

?: (hi) [How are you feeling?][Why are you here?]

These choices are picked equally. But maybe you don't want some choices. You can put an existence test of a variable at the start of a choice to restrict it.

?: (hi) [$ready How are you feeling?][Why are you here?]

In the above, the first choice is controlled by $ready. If it is undefined, the choice cannot be used. You can also use negative tests.

?: (hi) [!$ready this is a][This is b]

In the above only if $ready is undefined can you say *this is a*

If you want the variable to be a lead item in the actual output of a choice, you can do this:

?: (hi) [^eval($ready) is part of the output]

or the more clunky:

?: (hi) _0 = $ready [_0 is part of the output]

Choices lead to issues when you want rejoinders. You can label rejoinder branches of choices. Those without labels default to *a:*

?: (what meat) [c: rabbit] [e: steak] [h: lamb] [pork]

a: this rejoinders pork
c: this rejoinders rabbit
e: this rejoinders steak
f: this rejoinders on e:
h: this rejoinders lamb

In the above, *pork* rejoinders at *a:*, while the other choices name their rejoinder value. Each new starting label needs to be at least one higher than the rejoinder before it. That allows the system to detect rejoinders on rejoinders from choice branches.
If you do both variable control and rejoinder label, the control comes first and label after you have successful control.

?: (what meat) [$ready c: rabbit] [e: steak] [g: lamb] [pork]

Advanced Variables

Indirection Variables

You can, of course, merely refer to a variable in a script to set or get its value. But suppose you wanted to associate a variable with every topic you have. Maybe when the user says “I'm bored” you want to leave a topic and not reopen it yourself for any near future (say 200 volleys). In that case, you don't have a variable but you need to create one dynamically.

$$topicname = substitute(character %topic ~ “”)

The above gives you a topic name without the ~. Pretend the current topic is ~washing. Then you can create a dynamic variable name simply by using

$$tmp = ^join($ $$topicname)

which would create the name $washing.

Now suppose you wanted to set that variable to some number representing the turn after which you might return.

$$tmp = 25

doesn't work. It wipes out the $washing value of $$tmp and replaces it with 25.
You can set indirectly through $$tmp using function notation ^, e.g.

^$$tmp = 25

The above says take the value of $$tmp, treat it as the name of a variable, and assign into it. Which means it does the equivalent of

$washing = 25

If you want to an indirect value back, you can't do:

$$val = $$tmp

because that just passes the name of $washing over to $$val. Instead you do indirection again:

$$val = ^$$tmp # $$val becomes 25

Indirection works with values that are user variables and with values that are match variables or quoted match variables.

Many routines automatically evaluate a variable when using it as an argument, like ^gambit($$tmp). But if you want the value of the variable it represents, then you need to evaluate it another time.

$$tmp1 = ^$$tmp
^gambit($$tmp1)

Equivalently ^gambit(^$$tmp1) is legal.

See also Indirect Pattern Elements in Advanced Patterns.

Out of band Communication

ChatScript can neither see nor act, but it can interact with systems that do. The convention is that out-of-band information occurs at the start of input or output, and is encased in [ ].

ChatScript does not attempt to postag and parse any input sentence which begins with [and has a closing]. It will automatically not try to spellcheck that part or perform any kind of merge (date, number, propername). In fact, the […] will be split off into its own sentence. You can use normal CS rules to detect and react to incoming oob messaging. E.g, input like this

[speed=10 rate: 50] User said this

could be processed by your script. Although the 2 data oob items are inconsistently shown, the protocol you use is entirely up to you within the [] area.

OOB output needs to be first, which means probably delaying to one of the last things you do on the last sentence of the input, and using ^preprint(). E.g.

u: ($$outputgesture) ^preprint(\[$$outputgesture \])

You can hand author gestures directly on your outputs, but then you have to be certain you only output one sentence at a time from your chatbot (lest a gesture command get sandwiched between two output sentence). You also have to be willing to hand author the use of each gesture. I prefer to write patterns for common things (like shake head no or nod yes) and have the system automatically generate gestures during postprocessing on its own output.

The stand-alone engine and the WEBINTERFACE/BETTER scripts automatically handle the following oob outputs:

Callback: The webpage or stand-alone engine will wait for the designated milliseconds and if the user has not begun typing will send in the oob message [callback] to CS. If user begins typing before the timeout, the callback is cancelled.
e.g. [callback=3000] will wait 3 seconds.

Loopback: The webpage or stand-alone engine will wait for the designated milliseconds after *every* output from CS and if the user has not begun typing will send in the oob message [loopback] to CS. If user begins typing before the timeout, the loopback is cancelled for this output only, and will resume counting on the next output.
e.g. [loopback=3000] will wait 3 seconds after every output.

Alarm: The webpage or stand-alone engine will wait for the designated milliseconds and then send in the oob message [alarm] to CS. Input typing has no effect.
e.g. [alarm=3000] will wait 3 seconds and then send in the alarm.

CS can cancel any of these by sending an oob message with a milliseconds of 0.
e.g. [loopback=0 callback=0 alarm=0] cancels *any* pending callbacks into the future.

System callback functions:

outputmacro: ^CSBOOT()
This function, if defined by you, will be executed on startup of the ChatScript system. It is a way to dynamically add facts and user variables into the base system common to all users.

Note that when a user alters a system $variable, it will be refreshed back to its original value for each user.

Outputmacro: ^cs_topic_enter(^topic ^mode)
When the system begins a topic and this function is defined by you, it will be invoked before the topic is processed. You will be given the name of the topic and a character representing the way it is being invoked. Values of ^mode are: s, ?, u, t, which represent statements, questions, both, or gambits. While your function is executing, neither ^cs_topic_enter or ^cs_topic_exit will be invoked.

OutputMacro: ^cs_topic_exit(^topic ^result)
When the system exits a topic and this function is defined by you, it will be invoked after the topic is processed. You will be given the name of the topic and the text value representing what it returned. E.g., NOPROBLEM. The range of names of these are defined in mainsystem.h (minus _BIT) but are your basic FAILTOPIC, etc.

Advanced :build

Build warning messages

Build will warn you of a number of situations which, not illegal, might be mistakes. It has several messages about words it doesn't recognize being used as keywords in patterns and concepts. You can suppress those messages by augmenting the dictionary OR just telling the system not to tell you

:build 0 nospell

There is no problem with these words, presuming that you did in fact mean them and they do not represent a typo on your part.

You can get extra spellchecking, on your output words, with this:

:build 0 outputspell – run spellchecking on text output of rules (see if typos exist)

Build will also warn you about repeated keywords in a topic or concept. This means the same word is occurring under multiple forms. Again, the system will survive but it likely represents a waste of keywords. For example, if you write this:

topic: ~mytopic (cheese !cheese)

you contradict yourself. You request a word be a keyword and then say it shouldn't be. The system will not use this keyword. Or if you write this

topic: ~mytopic (cheese cheese~1)

You are saying the word cheese or the wordnet path of cheese meaning #1, which includes the word cheese. You don't need "cheese". Or consider:

topic: ~mytopic (cheese cheese~n)

Since you have accepted all forms of cheese, you don't need to name cheese~n.

:build also warns you about various substitutions that might affect your patterns. You can suppress those messages with ***:build filename nosubstitution***

Files

When you name a file or directory, :build will ignore files that end in ~ or .bak (the common backup names from editors on Windows and Linux).

Trace

Sometimes you might fail to place a paren properly, swallow a whole lot of input and crash. Finding where the problem is may be hard. You can therefore turn on a trace which will show you all the rules it successfully completes.

:build harry trace

Reset User-defined

Normally, a build will leave your current user identity alone. All state remains unchanged, except that topics you have changed will be reset for the bot (as though it has

not yet ever seen those topics). But if you want to start over with the new system as a new user, you can request this on the build command.
:build 0 reset – reinit the current character from scratch (equivalent to :reset user)

Build Layers

The build system has two layers, 0 and 1. When you say :build 0, the system looks in the top level directory for a file "files0.txt". Similarly when you say :build 1 it looks for "files1.txt". Whatever files are listed inside a "filesxxx.txt" are what gets built. And the last character of the file name (e.g., files0) is what is critical for deciding what level to build on. If the name ends in 0, it builds level 0. If it doesn't, it builds level 1. This means you can create a bunch of files to build things any way you want. You can imagine:
:build common0 – shared data-source (level 0)
:build george – george bot-specific (level 1)
:build henry – henry bot-specific (level 1)
    :build all – does george and henry and others (level 1)
or
    :build system0 - does ALL files, there is no level 1.

You can build layers in either order, and omit either.

Note- If you make something likes files2.txt and do a :build 2, this does not add another layer on top of level 1. It replaces level 1. There are only 2 levels. So if your file does not define a bot and his topics and control script, you wipe out Harry, for example, and are left with nothing.

Skipping a topic file

If you put in :quit as an item (like at the start of the file), then the rest of the file is skipped.

Block comments

Normally # becomes a comment to end of line. But you can use a block comment as follows:
    ##<< first junk
    some junk
    ##>> more junk
Because any comment marker kills the rest of the line, the "first junk" will not be seen, nor will the "more junk". But a comment block was established, so lines between them line "some junk" are also not seen.

Renaming Variables, Sets, and Integer Constants

A top level declaration in a script rename a match variable
    rename: _bettername _12
before any uses of _bettername, which now mean _12. You can put multiple rename pairs

in the same declaration.

rename: _bettername _12 _okname _14

and you can provide multiple names, so you can later also say

rename: _xname _12

and both _xname and _bettername refer to _12.

Renames can also rename concept sets:

rename: @myset @1

so you can do:

@myset += createfact(1 2 3)

$$tmp = first(@mysetsubject)

You can also declare your own 32 or 64-bit integer constants

Defining private Queries – see fact manual

Documenting variables, functions, factsets, and match variables

You can use :define to add a documentation string to many things. E.g.,

describe: $myvar "used to store data"

_10 "tracks pos tag"

:list can display documentation on documented items as well as showing undocumented permanent variables (handy for finalizing a bot to show you have no spelling errors on variables).

Editing Non-topic Files

Non-topic files include the contents of DICT and LIVEDATA.

DICT files

You may choose to edit the dictionary files. There are 3 kinds of files. The facts0.txt file contains hierarchy relationships in wordnet. You are unlikely to edit these. The dict.bin file is a compressed dictionary which is faster to read. If you edit the actual dictionary word files, then erase this file. It will regenerate anew when you run the system again, revised per your changes. The actual dictionary files themselves… you might add a word or alter the type data of a word. The type information is all in dictionarySystem.h

LIVEDATA files

The substitutions files consistof pairs of data per line. The first is what to match. Individual words are separated by underscores, and you can request sentence boundaries < and > . The output can be missing (delete the found phrase) or words separated by plus signs (substitute these words) or a %word which names a system flag to be set (and the input deleted). The output can also be prefixed with ![…] where inside the brackets are a list of words separated by spaces that must not follow this immediately. If one does, the match fails. You can also use > as a word, to mean that this is NOT at the end of the sentence. The files include:

interjections.txt – remaps to ~ words standing for interjections or discourse acts
contractions.txt – remaps contractions to full formatting
substitutes.txt – (omittable) remaps idioms to other phrases or deletes them.
british.txt – (omittable) converts british spelling to us
spellfix.txt – (omittable) converts a bunch of common misspellings to correct
texting.txt (omittable) converts common texting into normal english.
systemessentials.txt – things needed to handle end punctuation
expandabbreviations.txt – does what its name suggests

Processing done by various of these files can be suppressed by setting $cs_token differently. See *Control over Input.*

The queries.txt file defines queries available to ^query. A query is itself a script. See the file for more information.

The canonical.txt file is a list of words and override canonical values. When the word on the left is seen in raw input, the word on the right will be used as its canonical form.

The lowercasetitles.txt file is a list of lower-case words that can be accepted in a title. Normally lower case words would break up a title.

WHICH BOT?

The system can support multiple bots cohabiting the same engine. You can restrict topics to be available only to certain bots (see Advanced Topics). You can restrict rules to being available only to certain bots by using something like

?: ($bot=harry ...)
?: (!$bot=harry ...).
t: ($bot=harry) My name is harry.

The demo system has only one bot, Harry. If it had two bots in it, *harry* and *Georgia,* you could get Georgia by logging in as *yourname:georgia*. And you can confirm who she is by asking *what is your name*.

You specify which bot you want when you login, by appending *:botname* to your login name, e.g., bruce:harry . When you don't do that, you get the default bot. How does the system know what that is? It looks for a fact in the database whose subject will be the default bot name and whose verb is *defaultbot*. If none is found, the default bot is called *anonymous*, and probably nothing works at all. Defining the default bot is what a table does when you compile simplecontrol.top. It has:

table: defaultbot (^name)
^createfact(^name defaultbot defaultbot)
DATA:
harry

Typically when you build a level 1 topic base (e.g., :build ben or :build 1 or whatever), that layer has the initialization function for your bot(s) otherwise your bot cannot work. This function is invoked when a new user tries to speak to the bot and tells things like what topic to start the bot in and what code processes the users input. You need one of these functions for each bot, though the functions might be pure duplicates (or might not be). In the case of harry, the function is

^outputmacro: harry()
^addtopic(~introductions)
$cs_control_pre = ~control
$cs_control_main = ~control
$cs_control_post = ~control

SimpleControl.top also has a function that declares who the default bot is.

table: defaultbot (^name)
^createfact(^name defaultbot defaultbot)
DATA:
harry

You can change default bots merely by requesting different build files that name the default, or by editing your table.

To have multiple bots available, one might start by making multiple folder copies of Harry and changing them to separate bots with separate filesxxx.txt . That's fine for building them as each stand-alone bots, but if you want them to cohabit you must have a single filesxxx.txt file that names the separate bot's folders- they must be built together.

Of course if you merely cloned those folders, they have topics all with the same name, like ~control and ~introductions and ~childhood. This is a problem. Likely you would want only one copy of ~control that both bots used. And either they should have different topic names for ~introductions and ~childhood OR you must put a bot restriction on each of the duplicate topics, naming which bot can use it.

Topics By Bot

You should already know that a topic can be restricted to specific bots. Now you learn that you can create multiple copies of the same topic, so different bots can have different copies. These form a topic family with the same name. The rules are:

1. the topics share the union of keywords
2. :verify can only verify the first copy of a topic it is allowed access to

When the system is trying to access a topic, it will check the bot restrictions. If a topic fails, it will try the next duplicate in succession until it finds a copy that the bot is allowed to use. This means if *topic 1* is restricted to ben and *topic2* is unrestricted, ben will use *topic1* and all other bots will use *topic2*. If the order of declaration is flipped, then all bots including ben will use *topic 2* (which now precedes *topic 1* in declaration).

You can also use the :*disable* and :*enable* commands to turn off all or some topics for a personality.

Common Script Idioms

Selecting Specific Cases

To be efficient in rule processing, I often catch a lot of things in a rule and then refine it.

u: (~country) ^refine() # gets any reference to a country
a: (Turkey) I like Turkey
a: (Sweden) I like Sweden
a: (*) I've never been there.

Equivalently one could invoke a subtopic, though that makes it less obvious what is happening, unless you plan to share that subtopic among multiple responders.

u: (~country) ^respond(~subcountry)

topic: ~subcountry system[]
u: (Turkey) ...
u: (Sweden) ...
u: (*) ...

The subtopic approach makes sense in the context of writing quibbling code. The outside topic would fork based on major quibble choices, leaving the subtopic to have potentially hundreds of specific quibbles.

?: (<what) ^respond(~quibblewhat)
?: (<when) ^respond(~quibblewhen)
?: (<who) ^respond(~quibblewho)
... topic: ~quibblewho system []
?: (<who knows) The shadow knows
?: (<who can) I certainly can't.

Using ^reuse

To have a conversation, you want to volunteer information with a gambit line. And that same information may need to be given in response to a direct question by the user. ^reuse let's you share information.

t: HOUSE () I live in a small house
*u: (where * you * live) ^reuse(HOUSE)*

The rule on disabling a rule after use is that the rule that actually generates the output gets disabled. So the default behavior (if you don't set keep on the topic or the rule) is that if the question is asked first, it reuses HOUSE. Since we have given the answer, we don't want to repetitiously volunteer it, HOUSE gets disabled. But, if the user repetitiously asks the question (maybe he forgot the answer), we will answer it again because the responder didn't get disabled, just the gambit. And disabling applies to allowing a rule to try to match, not to what it does for output. So one can reuse that gambit's output any number of times. If you don't want that behavior you can either add a disable on the responder OR tell ^reuse to skip used rules by giving it a second argument (anything). So one way is:

t: HOUSE () I live in a small house

*u: SELF (where * you * live) ^disable(RULE SELF) ^reuse(HOUSE)*

and the other way is:

t: HOUSE () I live in a small house
*u: (where * you * live) ^reuse(HOUSE skip)*

Meanwhile, in the original example, if the gambit executes first, it disables itself, but the responder can still answer the question by saying it again.

Now, suppose you want to notice that you already told the user about the house so if he asks again you can say something like: *You forgot? I live in a small house.* How can you do that. One way to do that is to set a user variable from HOUSE and test it from the responder.

t: HOUSE () I live in a small house $house = 1
*u: (where * you * live) [$house You forgot?] ^reuse(HOUSE)*

If you wanted to do that a lot, you might make an outputmacro of it:

outputmacro: ^heforgot(^test) [^test You forgot?]
t: HOUSE () I live in a small house $house = 1
*u: (where * you * live) heforgot($house) ^reuse(HOUSE)*

Or you could do it on the gambit itself in one neat package.

outputmacro: ^heforgot(^test) [^test You forgot?] ^test = 1

t: HOUSE () heforgot($house) I live in a small house.
*u: (where * you * live) ^reuse(HOUSE)*

QUESTIONS/HAND-HOLDING

This is a reference manual, not a tutorial guide. Maybe you didn't see how to do what you wanted to do. Maybe it's possible at present and maybe it's not.

Feel free to email Bruce Wilcox at gowilcox@gmail.com and ask questions. If you find something you can't do, maybe I'll whip up a new release version in which it is possible.

Esoterica and Fine Detail

Being first to converse

Normally when you log in in stand-alone mode, this initiates a new conversation and the chatbot speaks first. If you prefix your login name with *, you get to speak first and this continues any prior conversation you may have had.

Prefix labeling in stand-alone mode

You can control the label put before the bot's output and the user's input prompt by setting variables $botprompt and $userprompt. I set them in the bot's initialization code, though you can dynamically change them. The values can be literal or a format string. The value is used as the prompt. Hence the following example:

$userprompt = ^"$login: >"
$botprompt = ^ "HARRY: "

The user prompt wants to use the user's login name so it is a format string, which is processed and stored on the user prompt variable. The botprompt wants to force a space at the end, so it also uses a format string to store on the bot prompt variable.

In color.tbl is there a reason that the color grey includes both building and ~building?

Yes. Rules often want to distinguish members of sets that have supplemental data from ones that don't. The set of *~musician* has extra table data, like what they did and doesn't include the word *musician* itself. Therefore a rule can match on *~musician* and know it has supplemental data available.

This is made clearer when the set is named something list *~xxxlist*. But the system evolved and is not consistent.

How are double-quoted strings handled?

First, note that you are not allowed strings that end in punctuation followed by a space. This string "I love you. " is illegal. There is no function adding that space serves.

String handling depends on the context. In input/pattern context, it means translate the string into an appropriately tokenized entity. Such context happens when a user types in such a string:

I liked "War and Peace"

It also happens as keywords in concepts:

concept: ~test["kick over"]

and in tables:

DATA:
"Paris, France"

and in patterns:

u: ("do you know" what)

In output context, it means print out this string with its double quotes literally. E.g.

u: (hello) "What say you? " # prints out "What say you? "

There are also the functional interpretations of strings; these are strings with ^ in front of them.

They don't make any sense on input or patterns or from a user, but they are handy in a table. They mean compile the string (format it suitable for output execution) and you can use the results of it in an ^eval call.

On the output side, a ^"string" means to interpret the contents inside the string as a format string, substituting any named variables with their content, preserving all internal spacing and punctuation, and stripping off the double quotes.

u: (test) ^"This $var is good." # if $var is kid the result is *This kid is good.*

What really happens on the output side of a rule?

Well, really, the system "evaluates" every token. Simple English words and punctuation always evaluate to themselves, and the results go into the output stream. Similarly, the value of a text string like "this is text" is itself, and so *"this is text"* shows up in the output stream. And the value of a concept set or topic name is itself.

System function calls have specific unique evaluations which affect the data of the system and/or add content into the output stream. User-defined macros are just script that resides external to the script being evaluated, so they are evaluated. Script constructs like IF, LOOP, assignment, and relational comparison affect the flow of control of the script but don't themselves put anything into the output stream when evaluated.

Whenever a variable is evaluated, its contents are evaluated and their result is put into the output stream. Variables include user variables, function argument variables, system variables, match variables, and factset variables.

For system variables, their values are always simple text, so that goes into the output stream. And match variables will usually have simple text, so they go into the output stream. But you can assign into match variables yourself, so really they can hold anything.

So what results from this:

u: (x)

$var2 = apples

$var1= join($ var2)

37

I like $var1

$var2 is set to *apples*. It stores the *name* (not the content) of $var2 on $var1 and then *I like* is printed out and then the content of $var1 is then evaluated, so $var2 gets evaluated, and the system prints out *apples*.

This evaluation during output is in contrast to the behavior on the pattern side where the goal is presence, absence, and failure. Naming a word means finding it in the sentence. Naming a concept/topic means finding a word which inherits from that concept either directly or indirectly. Naming a variable means seeing if that variable has a non-null value. Calling a function discards any output stream generated and aside from other side effects means did the function fail (return a fail code) or not.

How does the system tell a function call w/o ^ from English

If like is defined as an output macro and if you write:

t: I like (somewhat) ice

how does the system resolve this ambiguity? Here, white space actually matters. First, if the function is a builtin system function, it always uses that. So you can't write this:

t: I fail (sort of) at most things

When it is a user function, it looks to see if the (of the argument list is contiguous to the function name or spaced apart. Contiguous is treated as a function call and apart is treated as English. This is not done for built-ins because it's more likely you spaced it accidently than that you intended it to be English.

How should I go about creating a responder?

First you have to decide the topic it is in and insure the topic has appropriate keywords if needed.

Second, you need to create a sample sentence the rule is intended to match. You should make a #! comment of it. Then, the best thing is to type :prepare followed by your sentence. This will tell you how the system will tokenize it and what concepts it will trigger. This will help you decide what the structure of the pattern should be and how general you can make important keywords.

What really happens with rule erasure?

The system's default behavior is to erase rules that put output into the output stream, so they won't repeat themselves later. You can explicitly make a rule erase with ^erase() and not erase with ^keep() and you can make the topic not allow responders to erase with *keep* as a topic flag. So… if a rule generates output, it will try to erase itself. If a rule uses ^reuse(), then the rule that actually generated the output will be the called rule. If for some reason it cannot erase itself, then the erasure will rebound to the caller, who will try to erase himself. Similarly, if a rule uses ^refine(), the actual output will come from a rejoinder(). These can never erase themselves directly, so the erasure will again rebound to the caller.

Note that a topic declared *system* NEVER erases its rules, neither gambits nor responders, even if you put ^erase() on a rule.

How can I get the original input when I have a pattern like u: (~emogoodbye)

To get the original input, you need to do the following:

u: (~emogoodbye)
$$tmptoken = $cs_token
$cs_token = 0
^retry(SENTENCE)

and at the beginning of your main program you need a rule like this:

u: ($$tmptoken _)*
$cs_token = $$tmptoken

…. now that you have the original sentence, you decide what to do
…. maybe you had set a flag to know what you wanted to do

Control Flow

There is no GOTO in chatscript. There are only calls. Calls always return. They return with noproblem or end or fail. Respond/Gambit calls enter a new topic. Any end/fail TOPIC will terminate them and return to the caller. end(TOPIC) has no consequence to the caller. fail(TOPIC) degrades to a fail-rule and terminates the calling rule unless the call was wrapped in NOFAIL(). Nofail(TOPIC) and Nofail(RULE) are equivalent (because you can't get back a failed topic flag). a call to reuse does not enter a new topic context, so if the reuse calls end(topic), that returns to the calling rule, which has received an end(topic). If not wrapped in a nofail(TOPIC), then the calling rule terminates with end topic.

Meanwhile, generic output done before gambit/reuse/retry/respond will be forced to output so that if any of those fail, the output is still emitted. Otherwise, normally output generic waits until end of rule to be put out completely or cancelled completely if a fail happens.

Fail does not cancel output from a print or output already emitted. Each rule knows what output count exists at its start, and so when it ends it can tell if it generated output (suppressing further rules of the topic). fail(rule), when the rule has already generated output, does not stop further rules of the topic from being run. It thus allows more output than normal.

There is no GOTO in chatscript. There are only calls. Calls always return. They return with noproblem or end or fail. Respond/Gambit calls enter a new topic. Any end/fail TOPIC will terminate them and return to the caller. end(TOPIC) has no consequence to the caller. fail(TOPIC) degrades to a fail-rule and terminates the calling rule unless the call was wrapped in NOFAIL(). Nofail(TOPIC) and Nofail(RULE) are equivalent (because you can't get back a failed topic flag). a call to reuse does not enter a new topic context, so if the reuse calls end(topic), that returns to the calling rule, which has received an end(topic). If not wrapped in a nofail(TOPIC), then the calling rule terminates with end topic.

Meanwhile, generic output done before gambit/reuse/retry/respond will be forced to output so that if any of those fail, the output is still emitted. Otherwise, normally output generic waits until end of rule to be put out completely or cancelled completely if a fail

happens.

Fail does not cancel output from a print or output already emitted. Each rule knows what output count exists at its start, and so when it ends it can tell if it generated output (suppressing further rules of the topic). fail(rule), when the rule has already generated output, does not stop further rules of the topic from being run. It thus allows more output than normal.

Pattern Matching Anomolies

Normally you match words in a sentence. But the system sometimes merges multiple words into one, either as a proper name, or because some words are like that. For example “here and there” is a single word adverb. If you try to match “We go here and there about town” with

u: (here *) xxx*

you will succeed. The actual tokens are “we” “go” “here and there” “about” “town”. but the pattern matcher is allowed to peek some into composite words. When it does match, since the actual token is “here and there”, the position start is set to that word (e.g., position 3), and in order to allow to match other words later in the composite, the position end is set to the word before (e.g., position 2). This means if your pattern is

u: (here and there *) xxx*

it will match, by matching the same composite word 3 times in a row. The anomaly comes when you try to memorize matches. If your pattern is

u: (_ and _*) xxx*

then _0 is bound to words 1 & 2 “we go”, **and** matches “here and there”, and _1 matches the rest, “about town”. That is, the system will NOT position the position end before the composite word. If it did, _1 would be “here and there about town”. It’s not.

Also, if you try to memorize the match itself, you will get nothing because the system cannot represent a partial word. Hence

u: (_and *) xxx*

would memorize the empty word for _0.

If you don’t want something within a word to match your word, you can always quote it.

*u: (X * ‘and *) xxx*

does not match “here and there about town”.

The more interesting case comes when a composite is a member of a set. Suppose:

concept: ~myjunk (and)

u: (_~myjunk *) xxx*

What happens here? First, a match happens, because ~myjunk can match and inside the composite. Second memorization cannot do that, so you memorize the empty word. If you want to not match at all, you can write:

u: (_’~myjunk *) xxx*

In this case, the result is not allowed to match a partial word, and fails to match. However, given “ My brothers are rare.” and these:

concept: ~myfamily (brother)

u: (_'~ myfamily *) xxx*

the system will match and store _0 = brothers. Quoting a set merely means no partial matches are allowed. The system is still free to canonicalize the word, so brothers and brother both match. If you wanted to ONLY match brother, you could have quoted it in the concept definition.

concept: ~myfamily ('brother)

Blocking a topic from accidental access

There may be a topic you don't want code like ^gambit() to launch on its own, for example, a story. You can block a topic from accidental gambit access by starting it with

t: (!~) ^fail(topic)

If you are not already in this topic, it cannot start. Of course you need a way to start it. There are two. First, you can make a responder react (enabling the topic). E.g.,

u: (talk about bees) ^gambit(~)

If the topic were bees and locked from accidental start, when this responder matches, you are immediately within the topic, so the gambit request does not get blocked.

The other way to activate a topic is simply ^AddTopic(~bees). A topic being the current one on the pending topics list is the definition of "~". A matching responder adds the topic to that list but you can do it manually from outside and then just say ^gambit(~bees).

Self-Reflection

In addition to reasoning about user input, the system can reason about its own output. This is called reflection, being able to see into one's own workings.

Because the control script that runs a bot is just script and invokes various engine functions, it is easy to store notes about what happened. If you called ^rejoinder and it generated output (%response changed value) you know the bot made a reply from a rejoinder. Etc.

To manage things like automatic pronoun resolution, etc, you also want the chatbot to be able to read and process its own output with whatever scripts you want. The set of sentences the chatbot utters for a volley are automatically created as transient facts stored under the verb "chatoutput". The subject is a sentence uttered by the chatbot. The object is a fact triple of whatever value was stored as %why (default is "."), the name of the topic, and the offset of the rule within the topic.

You can prepare such a sentence just as the system does an ordinary line of input by calling ^analyze(value). This tokenizes the content, performs the usual parse and mark of concepts and gets you all ready to begin pattern matching using some topic. Generally I do this during the post-process phase, when we are done with all user input. Therefore,

```
t: ^query(direct_v ? chatoutput ? -1 ? @9 ) # get the sentences
loop()
{
        $$priorutter = ^last(@9subject)
        ^analyze($$priorutter) # prepare analysis of what chatbot said -
        respond(~SelfReflect)
}
```

Reflective information is available during main processing as well. You can set %why to be a value and that value will be associated with any output generated thereafter. E.g., %why = quibble.

The system also sets $cs_tokencontrol to results that happen from input processing.

A Fresh Build

You've been building and chatting and something isn't right but it's all confusing. Maybe you need a fresh build. Here is how to get a clean start.
0. Quit chatscript.
1. Empty the contents of your USER folder, but don't erase the folder.
This gets rid of old history in case you are having issues around things you've said before or used from the chatbot before.

2. Empty the contents of your TOPIC folder, but don't erase the folder.
This gets rid of any funny state of topic builds.

3. :build 0 - rebuild the common layer
4. :build xxx – whatever file you use for your personality layer
Probably all is good now. If not quit chatscript. Start up and try it now.

Command Line Options

Most of the command line options are documented with the server documentation, because they affect that. Here are general ones.

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.

buffer=15 - how many buffers to allocate for general use (12 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. 12 buffers is generally 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.

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 \r\n as needed). 0 is unlimited.

File options:

livedata=xxx – name relative or absolute path to your own private LIVEDATA folder. Do not add trailing / on this path. Recommended 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 /.

Other options:

trace – turn on all tracing.

redo – see documentation for :redo in debugging manual

userfacts=n limit 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.

userlog - Store a user-bot log in USERS directory (default).
nouserlog - Don't store a user-bot log.

login=xxxx - 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 runs :build on the filename as level0 and exits.
build1=filename runs :build on the filename as level1 and exits. Eg. ChatScript build0=files0.txt will rebuild the usual level 0.

Updating CS Versions Easily

ChatScript gets updated often on a regular basis. And you probably don't want to have to reintegrate your files and its every time. So here is what you can do.

Create a folder for your stuff: e.g. MYSTUFF. Within it put your folder that you normally keep in RAWDATA and your filesxxx.txt files normally at the top level of ChatScript. And if you have your own hacked version of files from LIVEDATA, put your folder there also.

Then create a folder within yours called ChatScript and put the current ChatScript contents within that. You can also create a batch file, probably within your folder, that does a "cd ChatScript" to be in the right directory, and then runs ChatScript with the following parameters:

ChatScript livedata=../LIVEDATA english=LIVEDATA/ENGLISH system=LIVEDATA/SYSTEM

Normally while you might override various substitutes files, you would not override the ENGLISH and SYSTEM folders.

So now, when you want to update to the latest version of ChatScript, merely unpack the zip into your ChatScript folder, overwriting files already there.

The Dictionary

There is the GLOBAL dictionary in DICT and local dictionary per level in TOPIC.
You can augment your dictionary locally by defining concepts with properties:

concept: ~morenouns NOUN NOUN_PROPER_SINGULAR (Potsdam Paris)
concept: ~verbaugment VERB VERB_INFINITIVE (swalk smeazle)

One can also directly edit the dictionary txt files in DICT/ENGLISH observing how they seem to be formatted, doing nothing crazy, and being careful with consistency of meaning values (if needed) and then just delete dict.bin. If you want to edit the wordnet ontology hierarchy, you need to edit facts.txt and delete facts.bin The system will rebuild them when you run CS.