ChatScript Advanced User's Manual
© Bruce Wilcox, gowilcox@gmail.com
Revision 12/31/2013 cs3.81

Table of Contents

ADVANCED CONCEPTS

Concepts can have part of speech information attached to them (using dictionary-system.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.

ADVANCED TOPICS

There are several things to know about advanced topics.

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

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,Georgia,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).

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.

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 cause others to generate output will not normally erase themselves:

```
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 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 semi-randomness 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.

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

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.

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.

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.

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

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

**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?
```

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?)

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 ~

You can ask if the system is currently in a topic via ~. This will consult the pending topics list, and if the topic is there, it is considered in that topic. 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).

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 Spelling

Some words you just know people will get wrong, like Schrodinger's cat or Sagittarius. You can request a partial match by specifying the initial letters, then using an * to represent the remaining letters. For example

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

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. 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.

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. Many routines automatically evaluate a variable when using it as an argument, like *^gambit($$tmp).* But some things don't, like *$$tmp = 1.* When you want a variable to hold another variable and indirectly access it, you can put a functional prefix on it. ^$ $tmp = 1 will evaluate the variable $$tmp and then try to assign 1 to whatever its value was. If $$tmp was $$y, then $$y = 1 will be the result of this.

Setting Match Position - @_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, @_2 makes the pattern think it just matched that wildcard at that location in the sentence.

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. The topic ~pronounanalyze below looks for male and female references in a sentence (only female is shown). When found, it uses a shared topic ~gathernounphrase to retrieve the complete reference.

```
topic: ~PRONOUNANALYZE system[]
u: ( _~she )
        $result = null
        respond(~GATHERNOUNPHRASE)( store

        if ($result) { $she = $result }
```

The topic ~GatherNounPhrase will try to locate a noun phrase starting from where the male or female reference matched in the above topic and then continuing to match backwards pieces.

```
topic: ~GATHERNOUNPHRASE system[]
u: (*) ^refine()  # select only one method of matching noun phrase
        # my mother is good
        a: ( @_0 _*-1 '_1?~determiner) $result = join('_1 _ _0)

        # is my aged mother good
        a: (  @_0 _*-2 '_1?~determiner _*1 _2=~adjective) $result = join('_1 _ _2 _ _0)

        # while my large aged mother is good, I'm not
        a: (  @_0 _*-3 '_1?~determiner _*1 _2=~adjective _*1 _3=~adjective)
                $result = join('_1 _ _2 _ _3 _ _0)

        # Rachel is good
        a: (  @_0 _0?~propername ) $result = _0
```

Reverse direction matching

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--.

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”]

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.

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).

Formatted double quotes

Programming languages allow you to control your output with format strings. 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. So

t: ^"I like you."

puts out *I like you*.

However, do not pass a format string as an argument to a function if it contains any local function variables. Instead, assign it to a temporary variable and pass that instead.

Function strings with local variables have to be handled in the context in which they are written and passing them as arguments to some function alters that context. So instead you evaluate them in their current context onto a temp variable, and pass the results of that evaluation around instead.

When you want special characters in the format string like [ ] and ( ), it is best to backslash them, which the format string removes when it executes. While the format string itself doesn't care, things around it might. 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 \[and \] .

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.

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

Loop Construct

The 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. Fail(rule) and End(rule) signal merely the end of the loop, not really the rule, and will not cancel any pending output in the output stream. All other return codes have their usual effect.

t: Hello. loop (5) { me }
t: loop () { This is forever, not. fail(RULE)}

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.

If Construct

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.

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.

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)

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.

System Functions

System functions are predefined and can be intermixed with direct output. You can write them with or without a ^ in front of their name. With is clearer, but you don't have to. The only time you must, is if the first thing you want to do in a gambit is call a function (unlikely).

t: name(xxxx) This is ambiguous. Is it function call or label and pattern?

The above is treated as a label and pattern. You can force it to be a function call by one of these:

t: ^name(xxx) # explicilty say it is a function
t: () name(xxx) # explicitly add an empty pattern

Topic Functions

^addtopic(topicname) – adds the named topic as a pending topic and clears the rejoinder marker. Typically you don't need to do this, because finding a reaction from a topic which is not a system, disabled, or nostay topic will automatically add the topic to the pending list. Never returns a fail code even if the topic name is bad.

^available(ruletag optionalfail) – sees if the named rule is available (1) or used up (0). If you supply the optional argument, the function will fail if the rule is not available.

^cleartopics() - empty the pending topics list.

^counttopic(topic what) – for the given topic, return how many rules match what. What is *gambit, available, rules, used.* That is, how many gambits exist, how many available gambits exist (not erased), how many top level rules (gambits + responders) exist, and how many top level rules have been erased.

^gambit(**value ?**) - If value is a topic name, runs the topic in gambit mode to see if any gambits arise. It does not fail unless a rule forces it to fail or the named topic doesn't exist

or is disabled. You can supply an optional 2nd argument FAIL, in which case it will return FAILRULE_BIT if it didn't fail but it didn't generate any new output either.

The value may be ~, which means use the current topic you are within. It can also be PENDING, which means pick a topic from the pending topics stack (they are all pending being returned to). Or it can be any other word, which will be a keyword of some topic to pick.

If the value is omitted entirely, the system will walk the pending topics list most recent first, trying each in turn to find a gambit. As it tries them it uses them up, so ones that fail to produce a gambit will disappear as pending topics for the future.

If the value is an ordinary word, the system will try to find topics with it as a keyword and execute those to get a gambit.

^getrule(what label) - for the given rule label or tag, return some fragment of the rule. What can be "tag", "type","label", "pattern", "output","topic", and "usable". The type will be t, ?, s, a, etc. If a rule label is involved, optional third argument if given means only find enabled rules with that label. For usable, returns 1 if is can be used or null if it has been erased.

^hasgambit(topic) – fails if topic does not have any gambits left unexecuted. Even it if does, they may not execute if they have patterns and they don't match. Optional second argument, if "any" will return normally if topic has any gambits (executed or not) and will failrule if topic has no gambits (a reactor topic).

^keep() - do not erase this top level rule when it executes its output part. (you could declare a topic to be this, although it wouldn't affect gambits). Doing keep() on a gambit is quite risky since gambits after it may not ever fire.

^lastused(topic,what) – given a topic name, get the volley of the last what, where what is GAMBIT, RESPONDER, REJOINDER, ANY. If it has never happened, the value is 0.

^next(what label) – given what of GAMBIT or RESPONDER or REJOINDER or RULE and a rule label or tag, find the next rule of that what. Fails if none is found. REJOINDER will fail if it reaches the next top level rule. If label is "~", it will use the last call's answer as the starting point, enabling you to walk rules in succession. There is also ^next(FACT @xxx) – see fact manual. For ^next(input) the system will read the next sentence and prep the system with it. This means that all patterns and code executing thereafter will be in the context of the next input sentence. That sentence is now used up, and will not be seen next when the current revised sentence finishes. Sample code might be:

```
t: Do you have any pets
    a: (~yes)  refine()
        b: (%more)  Next(input) refine()
            c: (~pets)  ….. react to pet
            c: () ^retry(SENTENCE)  # return to try input from scratch
```

b: () What kind do you have?
c: (~pets) …. react to pet

^**poptopic**(topicname) – removes the named topic as a pending topic. The intent is not to automatically return here in future conversation. If topicname is omitted, removes the current topic AND makes the current topic fail execution at this point.

^**refine**(?) - this is like a switch statement in C. It executes in order the rejoinders attached to its rule in sequence. When the pattern of one matches, it executes that output and is done, regardless of whether or not the output fails or generates nothing. It does not "fail", unless you add an optional **FAIL** argument. You can also provide a rule tag. Normally it uses the rule the refine is executing from, but you can direct it to refine from any rule.

^**rejoinder( ) –** see if the prior input ended with a potential rejoinder rule, and if so test it on the current sentence. If we match and dont fail on a rejoinder, the rejoinder is satisfied. If we fail to match on the 1st input sentence, the rejoinder remains in place for a second sentence. If that doesn't match, it is canceled. It is also canceled if output matching the first sentence sets a rejoinder.

^**respond**(topic-name ?) - tests the sentence against the named topic in responder mode to see if any rule matches (executes the rule when matched). It does not fail (though it may not generate any output), unless a rule forces it to fail or the topic requested does not exist or is disabled. This rule will not erase but the responding rule might. You can supply an optional 2nd argument FAIL, in which case it will return FAILRULE_BIT if it didn't fail but it didn't generate any new output either.

^**retry(item)** - if item is RULE rexecute the current rule. It will automatically try to match one word later than its first match previously. If item is TOPIC it will try the topic over again. If item is SENTENCE it will retry doing the sentence again. SENTENCE is particularly useful with ^Next(INPUT) since it won't have to reanalyze the sentence you have just brought in. ^retry(TOPRULE) will return back to the top level rule and retry. It's the same if the current rule was a top level rule, but if the current rule is from ^refine(), then it returns to the outermost rule to restart.

^**reuse**(rule-label optional-enable optional-FAIL) - uses the output script of another rule. The label can either be a simple rule label within the current topic, or it can be a dotted pair of a topic name and a label within that topic or it can be a rule tag. ^reuse stops at the first correctly labeled rule it can find and issues a RULE fail if it cannot find one. Assuming nothing fails, it will return 0 regardless of whether or not any output was generated.

When it executes the output of the other rule, *that* rule is credited with matching and is disabled if it is allowed. If not allowed, the calling rule will be disabled if it can be.

t: NAME () My name is Bob.
?: (<< what you name >>) ^reuse(NAME)
?: (<< what you girlfriend name >>) ^reuse(~SARAH.NAME)

Normally reuse will use the output of a rule whether or not the rule has been disabled. But…if you supply a 2nd argument (whatever it is), then it will ignore disabled ones and try to find one with the same label that is not disabled. You can also supply a FAIL argument (as either 2nd or 3rd) which indicates the system should issue a RULE FAIL if it doesn't generate any output.

There are also a variety of functions that return facts about a topic, but you have to read the facts manual to learn about them.

^**setrejoinder**(tag) – force the output rejoinder to be set to the given tag or rule label.

^topicflags(topic) – given a topic name, return the control bits for that topic. The bits are mapped in dictionary_system.h as TOPIC_*

Marking Functions

^mark (word match_variable1 match_variable2) – Marking and unmarking words and concepts is fundamental to the pattern matching mechanism, so the system provides both an automatic marking mechanism and manual override abilities. You can manually mark or unmark something. The effect of this is generally permanent for the rest of the sentence. You can also temporarily disable marks and then reenable them later.

^mark (~meat) – This marks ~meat as though it has been seen at whatever sentence start (location 1).
^mark (~meat _0) – This marks ~meat as though it has been seen at whatever sentence location _0 is bound to (start and end)
^mark (~meat n) – assuming n is within 1 and sentence word limit, this marks meat at nth word location.

Marking means the pattern matching system will react to it. Used typically to mark a topic to be visited when some idiomatic phrase implies it but keywords wouldn't. E.g.,

?: (what do you do) ^mark(~occupation)

Then you'd have a pattern within the topic to handle this idiom. Also for performing pronoun substitution. If the value of $she was mother, then

?: (she) ^mark($she)

would put the word mother marked as though it had been typed instead of she, so both she and mother occur in the same spot. Marking does not change the word, so a subsequent pattern

?: (_mother) would match from the mark, but _0 would be set to *she*.

^mark() – reestablish all transient marking saved by ^unmark(0)

When you are trying to analyze pieces of a sentence, you may want to have a pattern that finds a kind of word, notes information, then hides that kind of word and reanalyzes the input again looking for another of that ilk. Being able to temporarily hide marks and then reestablish them is what the ^mark(0) and ^mark(1) and ^unmark(* _1) are all about.

^marked(word) - returns "1" if word is marked, returns FAILRULE_BIT if the given word is not currently marked from the current sentence.

^unmark (word match-variable) – the inverse of ^mark, this takes a match-variable that was filled at the position in the sentence you want erased and removes the mark on the word or concept set or topic name given. Pattern matching for it in that position will now fail.

^unmark(* match-variable) is a special construction that says turn off ALL matches on this location temporarily. THEY are not overridden by a subsequent ^Mark to the same location. They expire when the volley ends or when you call ^unmark() to turn off all wildcard blocks of marks.

^unmark() memorizes the set of all * unmarks. You can restore them again with ^mark().

^position(how matchvariable) – this returns the integer representing where the named match variable is located. How can be START, END, or BOTH. Both means an encoding of where the start and end of the the match was. See @_n in pattern matchingto set a position or the ^setposition function.

^setposition(value) – sets the current match position to the numeric position in the sentence given by value. If value was returned as a "both" from ^position, then start and end are set to it. Otherwise start and end are both set to the simple number given. A "both" value is encoded as the lowest byte is the start and the next higher byte is the end.

Input Functions

^**capitalized(n) –** returns 1 if the nth word of the sentences starts with a capital letter in user input, else returns 0. If n is alphabetic, it returns whether or not it starts with a capital letter. Illegal values of n return failrule.

^**input(** …..) - the arguments, separated by spaces, are feed back into the input stream as the next input, injected before any pending additional input. Typically this command is then followed by ^fail(SENTENCE) to cancel current processing and move onto the revised input. Since the sentence is fed in immediately after the current input, if you want to feed in multiple sentences, you must reverse the order so the last sentence to be processed is submitted via input first.

^**position**(which _var)- If which is "start" this returns the starting index of the word matched in the named _var. If which is "end" this returns the ending index. E.g., if the value of _1 was "the fox", it might be that start was 3 and end was 4 in the sentence "it was the fox" .

^**removetokenflags(** value) - removes these flags from the tokenflags returned from the preprocessing stage.

^**settokenflags(value) -** adds these flags to the tokenflags return from the preprocessing stage. Particularly useful for setting the #QUESTIONMARK flag indicating the input was perceived to be a question. For example, I treat "tell me about cars" sentences as questions by marking them as such from script (equivalent to "what do you know about cars?".

Number Functions

^**compute**(number operator number) - performs arithmetic and puts the result into the output stream. Numbers can be integer or float and will convert appropriately. There are a range of operators that have synonyms, so you can pass in directly what the user wrote. The answer will be *?* if the operation makes no sense and *infinity* if you divide by 0. ~numberOperator recognizes these operations

+ plus add and (addition)
- minus subtract deduct (subtraction)

* x time multiply	(multiplication)
/ divide quotient	(float division)
% remainder modulo mod	(integer only- modulo)
root square_root	(square root)
^ power exponent	(exponent)
<< and >>	shift (limited to shifting 31 bits or less)
random	(0 random 7 means 0,1,2,3,4,5,6 - integer only)

Basic operations can be done directly in assignment statements like:

$var = $x + 43 - 28

^timefromseconds(seconds) – This converts time in seconds from the standard baseline in 1970, to a string like %time returns. You can compute a difference in times by merely doing a subtraction of the two times.

Output Functions

The following functions cannot be used during postprocessing since output has been finished in theory and you can now analyze it.

^flushoutput() - takes any current pending output stream data and sends it out. If the rule later fails, the output has been protected and will still go out (though the rule will not erase itself).

^insertprint (where stream) - the stream will be put into output, but it will be placed before output number where or before output issued by the topic named by where. The output is safe in that even if the rule later fails, this output will go out. Before the where, you may put in output control flags as either a simple value or a value list in parens.

^keephistory (who count) – The history of either BOT or USER (values of who) will be cut back to the count give. This affects detecting repeated input on the part of the user or detecting repeating output by the chatbot.

^print(stream) – sends the results of outputing that stream to the user. It is isolated from the normal output stream, and goes to the user whether or not one later generates a failure code from the rule. Before the output you may put in output control flags as either a simple value without a # (e.g., OUTPUT_EVALCODE) or a value list in parens. OUTPUT_EVALCODE is automatic, so not particularly useful. Useful ones would control how print decides to space things.

^preprint (stream **) -** the stream will be put into output, but it will be placed before all previously generated outputs instead of after, which is what usually happens. The output is safe in that even if the rule later fails, this output will go out. Before the output you may put in output control flags as either a simple value or a value list in parens.

^repeat() – allows this rule to generate output that may repeat what has been said recently by the chatbot.

^reviseOutput(n value)- allows you to replace a generated response with the given value. N is one based and must be within range of given responses. One can use this, for example, alter output to create accents. Using ^response to get an output, you can then use ^substitute to generate a revised one and put it back using this function.

Output Access

These functions allow you to find out what the chatbot has said and why.

^response(id) – what the chatbot said for this response.

^responsequestion(id) – Boolean 1 if response ended in ?, null otherwise

^responseruleid(id)- the rule tag generating this response from which you can get the topic.

PostProcessing Functions

These functions are only available during postprocessing.

^analyze(stream) – the stream generates output (not printed to user) and then prepares the content as though it were current input. This means the current sentence flagging and marking are all replaced by this one's. It does not affect any pending input still to be processed.

^postprocessprint(stream) – like preprint, it puts the stream first. You will not be able to analyze or retrieve information about this, like you would from a normal print.

Control Flow Functions

^command(args) – execute this stream of arguments through the **:** command processor. You can execute debugging commands through here. E.g.,

^command(:execute ^print(“Hello”))

^end(code) - takes 1 argument and returns a code which will stop processing. Any data pending in the output stream will be shipped to the user. If ^end is contained within the condition of an **if**, it merely stops it. An end rule inside a loop merely stops the loop. All other codes propogate past the loop. The codes are:

RULE – stops the current rule. Whether the next rule triggers depends upon whether or not output was generated.
TOPIC – stops the current topic.
SENTENCE - stops the current rule, topic, and sentence.
INPUT – stops all the way through all sentences of the current input.
PLAN – succeeds a plan – (only usable within a plan)

^eval (stream) – to evaluate a stream as though it were output (like to assign a variable). Can be used to execute :commands from script as well.

^**fail**(code) - takes 1 argument and returns a failure code which will stop processing. How extensive that stop is, depends on the code. If ^fail is contained within the condition of an **if**, it merely stops that and not anything broader. A fail or end rule inside a loop merely stops the loop; other forms propogate past the loop. The failure codes are:

RULE –stops the current rule and cancels pending output

TOPIC- stops not only the current rule also the current topic and cancels pending output. Rule processing stops for the topic, but as it exits, it passes up to the caller a downgraded fail(rule), so the caller can just continue executing other rules.

SENTENCE – stops the current rule, the current topic, and the current sentence and cancels pending output.

INPUT - stops processing anything more from this user's volley. Does not cancel pending output. It's the same as END(INPUT).

Output that has been recorded via ^print, ^preprint, etc is never canceled. Only pending output.

^match(what **) –** This does a pattern match using the contents of what (usually a variable reference). It fails if the match against current input fails.

^nofail(code …script…) – the antithesis of fail(). It takes a code and script, executes the script and removes all failure codes through the listed code. This is important when calling ^respond and ^gambit from a control script. You would want a control script to pass along codes at the sentence level, but if the respond call generated a fail-rule return, you don't want that to stop all the code of a control script responder.

^notnull(stream) - execute the stream and if it returns no text value whatsoever, fail this code. The text value is not used anywhere, just tested for existence. Useful in IF conditions.

Word Manipulation Functions

^burst(data-source burst-character) – takes the data source text and hunts within it for instances of the burst-character. If it is being dumped to the output stream then only the first piece is dumped. If it is being assigned to a fact set (like @2) then a series of transient facts are created for the pieces, with the piece as the subject and ^*burst* ^*burst* as the verb and object. If it is being assigned to a match variable, then pieces are assigned starting at that variable and moving on to successively higher ones. If burst_character is omitted, it is presumed to be _ (which separates words). If burst does not find a separator, it puts out the original value. For assignment to match variables, it also clears the next match variable so the end of the list will be a null match variable.

^explode (word) – convert a word into a series of facts of its letters.

^flags(word) – get the 64bit systemflags of a word

^intersectwords (arg1 arg2 optional) – given two "sentences", finds words in common in both of them. Output facts will go to the set assigned to, or @0 if not an assignment statement. The optional third argument, if "valuable", will only match words which are nouns, verbs, adjectives, or adverbs. Or if it's "canonical", it will match the canonical forms of each word.

^join (any number of arguments) – concatenates them all together, putting the result into the output stream. If the first argument is AUTOSPACE, it will put a single space between each of the joined arguments automatically.

^properties(word) returns the 64bit properties of a word

^pos(part-of-speech word supplemental-data) - generates a particular form of a word in any form and puts it in the output stream. If it cannot generate the request, it issues a RULE failure.

^tally (word {value}) Only valid during current volley. You can associate a 32-bit number with a word by ^tally(test 35) and retrieve it via ^tally(test).

^walkdictionary('function) calls the named output macro from every word in the dictionary. The function should have 1 argument, the word.

SYLLABLE – computes the number of syllables of the word. No 3rd argument.

TYPE – returns "concept" if arg2 starts with ~, returns "number" if it starts with a digit or + or -, returns "word" if it starts with a letter, and returns "unknown" otherwise.

NOUN x SINGULAR, PLURAL, PROPER, 1, other positive numbers
This converts the word to the appropriate designation.

NOUN x IRREGULAR
Generate a response only if the plural of x is irregular.

NOUN x lowercaseexist
this fails if the word, in lower case, has no dictionary entry. Often the system will have difficulty if a proper noun title of a book or film matches that of a simple lower case word. I generally omit titles that conflict, like "Love Story" conflicts with the simple love_story, a romance tale. This test allows you to avoid creating facts from risky names.

NOUN x uppercaseexist
this fails if the word, in upper case, has no dictionary entry.

VERB x PRESENT_PARTICIPLE, PAST_PARTICIPLE,INFINITIVE, PAST, MATCH

This outputs the requested form of x. For Match, the 3rd argument is a noun and the system will generate either the 3rd person form of the verb (for singular nouns) or the infinitive form of the verb (for plural nouns – unless the verb is "be").

AUX x pronoun
Generate correct form of auxiliary verb x, given personal pronoun.

PRONOUN x flip
Convert the given pronoun into the appropriate opposite (for I and you).

DETERMINER x –
This outputs the requested form of x. If x is a gerund, only x is output. If x already has an *a, an, the* as the head of the phrase, only x is output. If x is a mass noun or a proper noun, only x is output. If x is plural, *the x* is output, otherwise *a* or *an* is output.

PLACE n – generates appropriate place- 1st, 2nd, 3rd, 12th, etc. given n.

CAPITALIZE word – make the word start uppercase.

LOWERCASE word – make the word be lowercase.

CANONICAL word – returns the canonical form of a word.

INTEGER n – converts a float to an int by truncation.

^**rhyme**(word) – finds a word in the dictionary which is the same except for the first letter (a cheap rhyme).

^**substitute(** mode find oldtext newtext) – outputs the result of substitution. Mode can be **character** or **word**. In the text given by find, the system will search for oldtext and replace it with newtext, for all occurrences. This is non-recursive, so it does not also substitute within replaced text. Since *find* is a single argument, you pass a phrase or sentence by using underscores instead of spaces. ^substitute will convert all underscores to spaces before beginning substitution and will output the spaced results. In character mode, the system finds oldtext as characters anywhere in newtext. In word mode it only finds it as whole words in newtext.

^substitute(w "I love lovely flowers" love hate) outputs I hate lovely flowers
^substitute(c "I love lovely flowers" love hate) outputs I hate hately flowers

^**spell**(pattern fact-set) – given a pattern, find words from the dictionary that meets it and create facts for them that get stored in the referenced fact set. The facts are created with subject *1*, verb *word*, and object the found word. The pattern is a text string describing possibly the length and letter constraints. If there is an exact length of word, it must be first in the pattern. After which the system matches the letters you provide against the start of the word up until your pattern either ends or has an asterisk or a period. A period means match any letter. An asterisk matches any number of letters and would normally be

followed by more letters. The * will swallow letters in the dictionary word until it can match the rest of your given pattern. It will keep trying as needed. Eg.

^spell(4the @1) will find them but not their
*^spell(am*ic @1) will find American*
*^spell(a*ent @1) will find abasement*
^spell(h.l.o @1) will find hello

^sexed(word he-choice she-choice it-choice) – given a word, depending on its sex the system outputs one of the three sex choices given. An unrecognized word uses it.

^sexed(Georgina he she it) would return she

^uppercase(word) Is the given word starting with an uppercase letter? Match variable binds usually reflect how the user entered the word. This allows you to see what case they entered it in. Returns 1 if yes and 0 otherwise.

Dictionary Functions

^addproperty (word flag1 … flagn) – given the word, the dictionary entry for it is marked with additional properties, the flags given which must match property flags or system flags in dictionarySystem.h. Typically used to mark up titles of books and things when building world data. In particular, however, if you are adding phrases or words not in the dictionary which will be used as patterns in match, you should mark them with PATTERN_WORD. To create a dynamic concept, mark the set name as CONCEPT.

^define (word) – output the definition of the word. An optional second argument is the part of speech: noun verb adjective adverb, which will limit the definition to just that part of speech. Never fails but may return null.

^hasanyproperty(word value) – does this word have any of these property or systemflag bits? You can have up to 5 values as arguments, e.g., ^hasproperty(dog NOUN VERB ADJECTIVE ADVERB PREPOSTION).

^hasallproperty(word value) – does this word have all property or systemflag bits mentioned? You can have up to 5 values as arguments, e.g., ^hasallproperties(dog NOUN VERB ADJECTIVE ADVERB PREPOSTION). Values should be all upper case.

^removeproperty(word value) – remove this property bit from this word. This effect lasts until the system is reloaded. It is really only useful during the building of the dictionary itself. Value should be all upper case.

Multipurpose Functions

^disable (what ?) What can be “topic” or “rule” or “rejoinder”. If topic, the next argument can be a topic name (with or without ~). It means to disable (BLOCK) that topic. If a rule, you erase (disable) the labeled rule (or rule tag). If rejoinder, it cancels the current output rejoinder mark, allowing a new rule to set a rejoinder.

^enable (what ?) What can be “topic” or “rule”. If topic, the next argument can be a topic name or the word “all” for all topics. Designated topics will be enabld (unBlocked). If a rule, the label (or rule tag) will be enabled, allowing the rule to function again.

^length(what) If what is a fact set like @1, length returns how many facts are in the set. If what is a word, length counts its characters. If what is a concept set, length returns a count of the top level (nonrecursive) members.

^pick (what) – retrieve a random member of the concept if what is a concept. Pick is also used with factsets to pick a random fact (see FACTS MANUAL). For a concept, if the member chosen is itself a concept, the system will recurse to pick randomly from that concept. If the argument to pick is a $ or _var, it will be evaluated and then pick will be tried on the result (but it won't recurse to try that again).

^reset (what ?) – what can be *user* or *topic or factset*. If what is user, the system drops all history and starts the user afresh from first meeting, having erased the user topic file. If what is a factset, the “next” pointer for walking the set is reset back to the beginning. If what is a topic, all rules are re-enabled and all last accessed values are reset to 0.

Facts Functions – see FACTS MANUAL

^Iterator -

An iterator allows you to walk through each member of a concept, either at top level or recursively. Useful in conjunction with a loop(), the function is defined in the planning manual but can be used outside of planning. You can have one iterator in progress per rule.

An

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]

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 $token. The default one that comes with Harry is:

```
$token = #DO_INTERJECTION_SPLITTING  | #DO_SUBSTITUTE_SYSTEM   | #DO_NUMBER_MERGE
| #DO_PROPERNAME_MERGE  | #DO_SPELLCHECK | #DO_POSTAG  | #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:

```
DO_ESSENTIALS - perform LIVEDATA/systemessentials which mostly strips off trailing
punctuation and sets corresponding flags instead.
```

DO_SUBSTITUTES – perform LIVEDATA/substitutes
DO_CONTRACTIONS – perform LIVEDATA/contractions, expanding contractions.
DO_INTERJECTIONS – perform LIVEDATA/interjections, changing phrases to interjections.
DO_BRITISH – perform LIVEDATA/british, respelling brit words to American.
DO_SPELLING – performs the LIVEDATA/spelling file (manual spell correction)
DO_TEXTING – performs the LIVEDATA/texting file (expand texting notation)
DO_SUBSTITUTE_SYSTEM – do all LIVEDATA 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")

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:
DO_POSTAG – allow pos-tagging (labels like ~noun ~verb become marked)
DO_PARSE – allow parser (labels for word roles like ~main_subject)
DO_CONDITIONAL_POSTAG – perform pos-tagging only if all words are known. Avoids wasting time on foreign sentences in particular.

Normally the system tries to outguess the user, who cannot be trusted to use correct punctuation or casing or spelling. These block that:
STRICT_CASING – except for 1st word of a sentence, assume user uses correct casing on words.
NO_INFER_QUESTION – the system will not try to set the QUESTIONMARK flag if the user didn't input a ? and the structure of the input looks like a question.
DO_SPELLCHECK - perform internal spell checking
ONLY_LOWERCASE – force all input (except "I") to be lower case, refuse to recognize uppercase forms of anything

NO_IMPERATIVE -
NO_VERB -
NO_WITHIN -
NO_SENTENCE_END -

Normally the tokenizer breaks apart some kinds of sentences into two. These prevent that:
NO_HYPHEN_END – don't break apart a sentence after a hyphen
NO_COLON_END – don't break apart a sentence after a colon
NO_SEMICOLON_END – don't break apart a sentence after a semi-colon

Note, you can change $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 $token back to normal after you are done.

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.

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.

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 $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 actually has two bots in it, *harry* and *Georgia*. By default you get *harry.* You can 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. 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)
    $control_pre = ~control
    $control_main = ~control
    $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.

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.

Which Language?

The default language of the system is English. But if others create support for a different language, you can start up in any language and/or switch to any language.

By default the system starts up in English, but if you have a top level file called language.txt, it can name the startup language in it as the 1st and only line, e.g.,
german

In addition to loading a german dictionary instead, the system will also look for a top level parameter file "GERMAN.txt" (upper case for language name). If present, you can put command lines parameters, one per line, inside of it. You can use the same format as command lines:
hash=10325
buffers=20x15
or you can omit the = and use a space instead.

Normally authors don't switch back and forth between languages on the fly. Particularly because the TOPIC folder will have been written generally to use only one language at a time. However, typing :restart german will request a system restart with that language, and write out a new language.txt file with that as the default language of the future.

In some cases, you may have a dictionary which is different from the standard. E.g., Angela (Outfit7) had a miniaturized dictionary (fewer words). That is merely another language name, with a folder in DICT for the corresponding "language". Similarly one could then have an ANGELA.txt parameters file.

Reading Documents

Normally the system reads a line of input from the user and responds to it, completing a volley. But there is another style of input available, called document mode. The :document command is given the name of a document or directory of documents.
:document myfile.txt
:document mydirectory/

The document will automatically be broken into sentences and fed to the system one at a time. The system will execute the prepass once at start, and the postpass once at end. All sentences of the document are treated as a single volley, so the user topic file is not updated until after the document has been fully read.

Optional arguments after the file/directory name are:
~topicname - a postprocessing topic to invoke after all files have been processed.
echo – the lines read will be dumped to TMP/out.txt so you can see what it read.
s*ingle* – read a single line at a time, even if it is not obviously a complete sentence.

%document is a system variable you can use to tell if you are in document mode or not, to make your preprocess and postprocess topics act conditionally on whether you are or not. And the current document path is stored on the variable $$document.

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

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 functional 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 it is evaluated. Script constructs like IF, LOOP, assignment, and relational comparison affect the flow of control of the script but don't put anything themselves into the output stream when evaluated.

Whenever a variable is evaluated, it doesn't put its contents into the output stream--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. You can assign into match variables yourself, so really they can hold anything.

So what result s from this:

```
u: (x)
    $var2 = apples
    $var1= join($ var2)
    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 % 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.

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 = $token
        $token = 0
        ^retry(SENTENCE)
```

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

```
u: ($$tmptoken _* )
        $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

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 you 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: (* '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 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 ^gambits(~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

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

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.

livedata=xxx – name relative 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 also have a top level file, e.g., ENGLISH.txt with parameters in it. See What Language?

Other options:

trace – turn on all tracing.

language – set the language to this. Normally the system defaults to english. You can (if implemented) pick other languages.

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).

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)