ChatScript User's Manual

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

Revision 6-11-2011

Table of Contents

OVERVIEW

ChatScript is a scripting language designed to accept user text input and generate a text response. Chat proceeds in volleys, like tennis. The program inputs one or more sentences from the user and outputs one or more sentences back.

Rules: Fundamentally a script is a series of rules. A full rule has a kind, a label, a pattern, and an output. Here is a simple full rule:

?: MEAT (you like meat) I do.

The rule kind is *?:*, which means it only reacts to questions. The label is MEAT. The pattern is in () and looks to find the words *you like meat* in consecutive order, anywhere in the input, e.g. *do you like meat.* The output is *I do.*

The kind restricts when the rule can be tried. You can restrict rules to statement inputs, question inputs, or only when the bot takes control of the conversation and wants to volunteer something (gambits). Kinds are a letter follow by a colon.

The label is optional and is partly a documentation and debugging aid and allows this rule to be manipulated by other rules.

A pattern is a set of more specific conditions which allow or disallow this rule, usually trying to match words of the current input sentence, but sometimes taking into account the prior history of the conversation, the time of day, or whatever.

The output is what this rule does if allowed to execute. Since the goal of the overall system is to generate a response, the simplest output is merely the words to say. More complex outputs can do conditional execution, loops, function calls, etc.

The system normally executes rules in a specified order until one not only passes the kind and pattern restrictions, but also actually generates output destined to reach the user.

Once one has output, the system is done (unless you explicitly want to do more).

Topics: Rules are bundled into collections called topics. You tell the system to execute a topic, and it begins executing rules in that topic until it generates an output. Topics can invoke other topics. Exactly how to process an input is controlled by a control script which itself is merely a topic. It generally calls engine functions and conditionally executes other topics.

Whitespace: ChatScript generally ignores excessive whitespace. You can have a plethora of tabs, spaces, and newlines. But, you do have to use whitespace to separate tokens.

Case: ChatScript is case insensitive for code script. Obviously case is important in literal output. And normally words in patterns should be lower case unless they are proper names, but the system can manage that if you do it wrong.

***Comments*:** the comment character is the hash mark. It extends to the end of the line.

s: (I hate meat) So do I. # an untruth

Hello Word

Let's turn to running the system in a simple demo.

1. Download and extract on a windows system into a directory (mine is called ChatScript), keeping files in their respective folders.
2. Double click on the chatscript.exe file. The system prints out a bunch of statistics as it loads data. After which is says: "Enter user name".

- if you are on LINUX, run LinuxChatScript with argument "local"

3. Select a user name. It's arbitrary, but it's how the system knows you are you when you start up again later. The system will respond with a *Welcome to ChatScript* message.
4. Now enter: *What is your name?* – you get *My name is Harry* if you type this correctly. You can also ask *How was your childhood?* and *what are you afraid of* and *what is your history*

OK. Now you've run what comes pre-installed. Let's quickly survey what comes built in.

Simple Editing the Script

To change the script, use some word editor (WORD, notepad, whatever) and open up RAWDATA/simpleTopic.top. In it find the line at the bottom that says *My name is Harry* and change it to *My name is Rachel.* Then, while running the ChatScript engine, (having provided a user name), enter *:build 1*. This rebuilds the simple script and reinitates a conversation. You can then type in *What is your name?* and get back your new answer. The system still considers the persona running to be labeled Harry, but that's just its internal label.

If you want to create new topics, you should read the Simple Topics documentation. In RAWDATA/skeleton.top there are a whole slew of predefined topic declarations with keywords. You can either add this file name into your files1.txt and then edit some topic or copy a specific topic into some other file and reference that from files1.txt (including the already referenced simpleTopic.top file).

Simple Modification

OK. Let's look at actually changing the system.

In the RAWDATA folder are two top-level files. simpleControl.top is your control script. simpleTopic.top contains your main script for things you want the bot to say. These two files are listed in the top level file *files1.txt* .

When you make changes to one of these *.top* files, you compile them by typing ***:build 1*** to the chatbot. This replaces some content in the TOPIC directory (and maybe the VERIFY directory) and reloads all files so you can immediately try out your changes. Because you can have a lot of files and data, the system actually supports two stages of building data. Things that you don't expect to change often should be in files listed in *files0.txt* and are rebuilt using ***:build 0.*** Things you are actively changing a lot should go into files of *files1.txt* and get rebuilt with ***:build 1.***

Fast Overview of SimpleTopic.top file

Chatscript is so easy, a child could master it in seconds. And if you believe that, I have several bridges available for sale to you on special. But some people will try to dive right in, without reading the material, so here is a quick guide to what you see in this simple topic file.

Rules start with t: or ?: or u: or s:

    s: means the rule reacts to statements.
    ?: means the rule reacts to questions.
    u: means the rule reacts to the union of both.
    t: means the rule offers a topic gambit when chatbot has control

Rules also start with a: b: etc, but those are "rejoinders", not top level. They are used to anticipate how a user might respond to output and give direct feedback based on that response.

Rules are thus classified as:

Responders (s: ?: u:) which are rules that try to react to unprovoked input from the user. That is, he might out of the blue ask you something or say something, and these attempt to cope with that.

Rejoinders (a: b: … q:) are attempts to predict a user's immediate response to something the chatbot says. They cannot be triggered except on input immediately after the rule they follow has issued output.

Gambits (t:) are the story the chatbot wants to tell on a subject or the conversation the chatbot is trying to steer the user into.

Rules usually have pattern requirements in parens (except t: rules for which a pattern is optional). These typically try to find specific words or sequences of words in the user's input. In the rule:

    u: (run away)

the engine will see if the word *run* can be found anywhere in the sentence, and if so, is it immediately followed by the word *away*.

Whenever you see […] it means pick one of. So

    u: ([scare afraid])

means find anywhere in the sentence *one of* the words *scare* or *afraid.* And *scare* can be in any of its related forms: *scared, scare, scaring, scares*.

In the sample file you will see ordinary words and ~words. Tilde words refer to a concept set of words, a list of words that approximates the ~word. E.g., *~like* means any of a number of words that mean to like something. *~animals* means any of a large list of names of animals. These are shareable shorthand for the [ ] notation. Instead of having to write *[elephant tiger leopard alligator crocodile lion ....]* in lots of rules, with the appropriate concept defined one can merely say *~animals*

Rules are always bundled into topics, like the topic: ~childhood topic. Topics have a list of relevant keywords following the topic name. After that, the *gambits*, t: lines are always the first rules in a topic. They offer a story or expected conversation flow. If you ask information in a conversation, you are expected to balance the scales by giving information. For example if you ask what someone does for a hobby, you are expected after their response to answer the question about yourself. As in:

> *Topic: school [school university learn]*
> *t: Where do you go to school?*
> *t: I go to Harvard?*
> *t: What is your major?*
> *t: I am studying finance.*

Of course one is often expected to respond to the user's response. So if he answers "where do you go to school" by saying "the university of Rochester", it helps if you can make some cogent *rejoinder* on that BEFORE you gambit say "I go to Harvard".

And your topic has to be ready to handle arbitrary school-related questions from the user. If he says "I go to Yale. What was your school mascot?", you'd like to have added *responders* that could field the mascot question before you move on to volunteering you went to Harvard.

This is what the simple topic on childhood attempts to do. Ask gambit questions, respond with appropriate remarks to their response, offer the chatbot's answer to the gambit, move on, and handle some simple questions asked out of the blue on the topic.

Comments start with # . In the file, the comment *# issued only once*, is an ordinary comment.

Special comments #! give sample input from a user that the immediately following rule is expected to match and handle. This both documents what input is expected to match the rule below AND allows the engine to automatically test it. The special comment gives only 1 example of matching input, not all possible inputs that can match. They help you understand what a responder or rejoinder is supposed to react to. They have no impact whatsoever on a user in chat.

Overview of the control script

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).

What's Where Overview

The ChatScript engine can run multiple bots at once, each with a unique persona. So one user can connect and talk with a specific personality while another user connects and talks with a different one (or the same one).

History Files: Each user's conversation is tracked by the system and kept in files in the USERS directory. A user can return to chat with a personality days later, and the system knows what has happened in previous conversations and that this is the start of a new conversation. The system keeps a log file per user recording their conversations for the author, and a topic file for each user-chatbot pairing, where it stores the current state of conversation for the engine. If the script records facts about the user during conversation, these are stored in a facts file, one per user, to be read in again by the engine.

Dictionary Files: The DICT folder is the underlying dictionary of the system. You probably won't modify it.

Dictionary Extension Files: The LIVEDATA folder contains extensions to the dictionary and runtime system that you might change as an advanced author.

Knowledge Files: The RAWDATA folder is where raw data to support chat is kept (though you can keep it anywhere since it's not compiled into the engine). That data is run through the script "compiler" and the output is stored in the TOPIC directory, which holds your compiled script data. If your script has verification data embedded in it, which allows the system to prove your patterns actually do what you intend, that data is stored in the VERIFY directory after compilation.

Source Files: src is source for rebuilding the engine. It has a file *dictionarySystem.h* which is read during loading to define engine constants that can be used in scripting.

Documentation Files: This document and others can be found in DOCUMENTATION.

Compilation Files: The folders LINUX, VS2008, and VS2010 are for rebuilding the executable. LOEBNERVS2008 and LOEBNERVS2010 build a Loebner version.

Top Level Files: Aside from the chatscript.exe, the following top level files exist:

authorizedIP.txt –as a server, this allows some users to enter commands
files0.txt & files1.txt – what data files to compile with :build 0 and :build 1
server.bat – run the engine as a windows server
LinuxChatScript – Linux executable engine (defaults to server mode)
loebner.exe- engine compatible with Loebner competition protocol
changes.txt – list of changes between releases
LocalClient.bat – file to run engine as a client on same machine as server

SIMPLE TOPICS

The system does not execute all rules. Nor do you directly choose a rule to execute. Instead, the author organizes collections of rules into topics and chooses what topic to execute at any moment. It, in turn, executes its rules. Here is an example of a simple topic declaration.

> *topic: ~DEATH [dead corpse death die body]*
> *t: I don't want to die*
> *?: (When will you die) I don't know.*

The topic declares its name, its keywords, and then its rules. It ends with the end of the file or a new top level declaration (which includes *topic:*, *concept:*, *table:*, *outputmacro:*, *patternmacro:*). A topic name must start with a ~ .

Keywords

Keywords allow the system to consider this topic based on consanguinity with the input sentence. User sentence keywords trigger the closest matching topic (based on number and length of keywords) for evaluation. If the engine can find a matching responder, then it shifts to that topic and replies. Otherwise the engine tries other matching topics. Eventually, if it can't find an appropriate responder, it can go back to the most relevant topic and just dish out a gambit (the t: rule).

It doesn't matter if the new topic has responders that overlap some other topic's responders (both could match the input). You can have a topic on *~burial_customs* and another on *~death*. An input sentence *I don't believe in death* might route you to either topic, but that's reasonable.

Topics make it easy to bundle rules logically together. Topic keywords mean the author can script an independent area of conversation without regard to pre-existing script and the system will automatically find it and invoke it as appropriate.

Topics do not have to have keywords. Catch-all topics can be created for various purposes which have no keywords – they have the parens but no words within.

Gambit Rules

A topic introduces an additional rule type. In addition to responders for user input (*s: ?: u:*) it has topic gambits it can offer (*t:*). Gambits create a coherent story on the topic if the bot is in control of the conversation. Yet if the user asks questions, the system can use a responder to either respond directly or use up a gambit it was going to volunteer anyway. **All gambit rules must precede all responders**. It is entirely up to you the order of responders after the gambits. You can segregate *s:* from *?:* from *u:* or co-mingle them.

The typical gambit does not have a label or a pattern component. It could have them. It just usually is the rule type and the output data.

Execution Order

A topic is executed in either gambit mode (meaning *t:* lines fire) or in responder mode (meaning *s: ?:* and *u:* fire). Rules are placed in the order you want them to be tried. For gambits, the order tells a story. For responders, rules are usually ordered most specific to least specific, possibly bunched by subtopic. So a responder trying to catch *what color is your hair* would be before one that simply would react to any reference to *your hair*.

By default, the system avoids repeating itself, marking as used-up rules that it matches that generate output. This is how a topic story gets told. It outputs the first gambit, marks it used, and then next time it will output the second gambit and mark it used, and so on.

The file RAWDATA/skeleton.top has a bunch of topics already predefined with keywords but no responders or gambits. If you filled in some of these topics with rules and hooked the file into files1.txt and rebuilt the data, you'd have a chatbot.

Rejoinders

If you expect the user might respond in a particular manner to the chatbot's last output, you can script rules to examine his next input and see if it matches. When it works, it makes your chatbot seem like it understands the user. These are called rejoinder rules and all rules can have them.

```
s: ( I like spinach ) Are you a fan of the Popeye cartoons?
        a: ( yes ) I used to watch him as a child. Did you lust after Olive Oyl?
                b: ( no ) Me neither. She was too skinny.
                b: ( yes ) You probably like skinny models.
        a: ( no )  What cartoons do you watch?
                b: ( none ) You lead a deprived life.
                b: ( Mickey Mouse ) The Disney icon.
```

Rejoinders use *a:* through *q:* to indicate nesting depth. All rejoinders at a level have the same letter and are alternatives that will be tested in order. So, after the chatbot asks *are you a fan…* it will test the next input for *yes* and then *no*. As soon as it finds a matching rejoinder, it will execute it and be done. If it finds none, then the system just moves on to its normal behavior. Rejoinders can have rejoinders, as shown above. Indenting like above is good style, making it visually obvious in your script.

Note—technically the above uses of *yes* and *no* will not actually work as written. They are considered special and treated as interjections (along with many other things that mean the same thing). To make the above example actually work in the engine, you'd have to use *~yes* and *~no*. But you don't learn about concepts and interjections until later.

Rule Labels

All rules (responders, gambits, rejoinders) can have labels on them. Labels have a variety of uses. Other rules can use functions that target a particular labeled rule. You can use the debug abilities to test that rule and you can see that rule more easily in a trace. And you get a kind of documentation telling you what your rule is about. A label is a single word placed between the rule type and the pattern. If the rule is a gambit, you must add a pattern, even if it is only empty parens.

```
t: MY_EYES ()  My eyes are blue
?: EYECOLOR  (color * eyes) I have blue eyes
u: GLASSES ([glasses contacts]) I don't wear glasses, but I do have contacts.
?: BLIND (you * blind) I am not blind.
?: COLORBLIND  (you * [color-blind "color blind"])  I am not color blind.
```

The simpletopic.top file has an example topic called ~***Childhood*** of normal complexity (which can be understood after reading through advanced output).

SIMPLE PATTERNS

As stated previously, a rule cannot fire unless its pattern matches, and a pattern in a rule is encased in parens (which means find the items within it in sequence).

Writing patterns is a delicate balancing act. If you are too specific, the pattern will miss all sorts of opportunities to respond to similar meanings. If your pattern is
?: (when will you go home) I go home tomorrow
and the input is *when will you be going home*, the bot fails to react. But if your pattern is too broad, the bot responds to completely wrong meanings. If your pattern is
s: (home) I go home tomorrow.
then it reacts to *He slid home* inappropriately.

The goal, usually, is to write a pattern that matches a particular meaning. Before you can perform the balancing act, you need to see what things can be in a pattern.

In sequence ( )

I said that parens mean in sequence, anywhere in the input. Thus
s: (I love you) Do you really?
matches *How I love you!* and *I love you and your kind* and *Everyone knows I love you.*

You can even nest parens within parens, not that it has any functional utility.
s: (I (love you)) Do you really?
This pattern is equivalent to the earlier one without nested parens. Whereas the outer parens can start their first element matching anywhere in the sentence, once a positional context has been established, that gets inherited. Thus after *I* is matched, the starting context of the inner opening paren is that the next element must match in position 2 in the sentence, immediately after *I.*

Another way to request a sequence is to put double quotes around it. They must be normal ascii double-quotes, not Word's fancy left and right double quotes.
s: ("I love you") Do you really?
There are two reasons to use double quoting. First, if you are trying to shoe-horn a phrase into a place that expects a word. For example, as a topic keyword you could do this:
topic: ~death ["to die" "cross over"]
Second, when trying to write words where you are not sure how the system will tokenize it and whether it is one word or a sequence of words. Tokenization involving punctuation can be tricky. For example, the word *Bob's* is actually tokenized as two words: *Bob 's* . And in Wordnet, *New_Year's_Eve* is a single word. You might not know that, but anytime you think of something as multiple words, you are safe quoting it (writing "*New Year's Eve"*) and letting ChatScript manage how it is stored internally. This is particularly true of names of people, titles of books and other multiple-word proper names. Put things with punctuation in them in double quotes to be safe.

Sentence boundaries < and >

Sometimes, to get a proper meaning in the pattern, you need to actually know where an input begins or ends. For example:

u*: (what is an elephant) An elephant is a pachyderm.*

matches *Tell me what is an elephant* and *what is an elephant* and *what is an elephant doing in the room*. That last one is inappropriately matched.

The > matches the end of the sentence. This makes it possible to correctly manage the above sentences as follows:

u*: (what is an elephant >) An elephant is a pachyderm.*

The < doesn't really match the start of the sentence so much as it sets the current position of matching to the start of the sentence. Thus

u*: (roses < I like) I like roses too.*

matches *I like roses* because it finds *roses* anywhere in the sentence, then the < resets the match position to the sentence start, and then it finds *I like* at the beginning. Of course this will not match *You know I like roses* because *I* is not at the start of the sentence.

Simple Indefinite Wildcards *

The wildcard * means 0 or more words in sequence. It can be used to widen a pattern:

*?: (when * you * home) I go home tomorrow*

This pattern responds to *When will you go home* and *When Roger is with you, will there be anyone at home?*

Precise Wildcards *1 …

As you may notice, indefinite wildcards can allow all sorts of mischief to creep into a match. An overprotective way to manage this is using wildcards that tell you exactly how many words can be swallowed up. The * followed by a number names how many words it absorbs.

*?: (when *1 you *1 home) I went home yesterday*

This matches *When did you go home* but won't accept wide variances like *When Roger is with you…*nor will it accept *when you went home* which hasn't room for the first *1.

Range-restricted Wildcards *~1 …

The usual way to manage the excesses of the previous wildcards is to use a range-restricted wildcard. This is an * followed by a ~ and a number, like *~3. It means from 0 up through that number, or approximately that number. A common choice is *~2. This leaves room for some filler words (like a determiner and an adjective or perhaps some kind of adverb), without requiring them or letting the sentence stray.

*?: (you *~2 go *~2 home) I often go to that home.*

This responds equally to *You can go home* and *you should not go to your home.*

Unordered Matching << >>

Often times you are interested in matching several keywords, but you explicitly want any order of them. For example the sentence *I love birds* is a lot like *Birds are what I love* but subject and object move around. One somewhat tedious way to match in any order is:

*s: (I < * love < * birds) I love birds too.*

This works by going back to the beginning of the sentence and allowing any number of words to match a wildcard until the next keyword is found. It's ugly. The cleaner way is to use the unordered markers.

s: (<< I birds love >>) I love birds too.

Choices [ ]

You can match alternate words in the same position by placing those choices in brackets.

?: (you [swim ride fish]) I do.

This matches *Do you swim* and *Do you fish* and *do you ride*.

Choices may be significant alternatives or they can be synonyms.

*?: (you [eat ingest "binge and purge" (feed my face)] *~2 meat) I love meat*

Notice that elements of a choice can be sequences of words either as double-quoted phrases or as paren sequences.

~Concepts

Choices are handy for synonyms, but you have to repeat them over and over in different rules. At such point being able to declare a list of choices in one place and use them everywhere else becomes convenient. This is the concept set. It is hugely important in writing patterns that match meaning.

concept: ~eat [eat ingest "binge and purge"]

Unlike choices, a concept cannot use paren notation to hold a sequence of words, though it can use quoted expressions.

A concept is a top-level declaration consisting of a name starting with ~ and a list of words. You can use the set name in any pattern or topic keyword list in place of a word.

s: (I ~eat meat) Do you really? I am a vegan.

Think of the ~ as meaning approximately. Topic names are also concept names, with the keywords of the topic being the choices.

ChatScript can represent word synonyms as above or affiliated words as below.

concept: ~baseball [strike umpire ball bat base]
… some topic declaration
s: (~baseball) I'm not that into sports like baseball.

A concept can also a natural ordering of words that an advanced script can use. The ordered concept below shows the start of hand ordering in poker.

concept: ~pokerhand [royal flush straight flush 4 of a kind full house]

The pattern:

*?: (which * better * ~pokerhand * or * ~pokerhand) ...*

detects questions like *which is better, a full house or a royal flush* and the system has functions that can exploit the ordered concept to provide a correct answer.

You can nest concepts within concepts, so this is fine:

concept: ~food [~meat ~dessert lasagna ~vegetables ~fruit]

Hierarchical inheritance is important in pattern generalization. Concepts can be used to create full ontologies of verbs, nouns, adjectives, and adverbs, allowing one to match general or idiomatic meanings. The system comes with such already defined, you just have to activate it. If you give the command **:build 0** to the chatbot, you will build the underlying ontology and world knowledge of the system. Then you can explore the existing sets.

In addition to fixed sets (over 1300 of them), the system automatically defines a bunch of dictionary-based sets. These include:

parts-of-speech like *~noun* (see POS-Tagging)
~number and *~placenumber* (like 1st, second)
~propername, ~malename, ~femalename, ~humanname, ~firstname
~childword, ~adultword, ~unknownword
~timeword (refers to time), *~locationword* (refers to a place), ~url

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.

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.

Interjections, “discourse acts”, and concept sets

Some words and phrases have interpretations based on whether they are at sentence start or not. E.g., *good day, mate* and *It is a good day* are different for “good day”. Likewise *sure* and *I am sure* are different. Words that have a different meaning at the start of a sentence are commonly called interjections. In ChatScript these are defined by the *livedata/subtitutes.txt* file. In addition, the file augments this concept with “discourse acts”, phrases that are like an interjection. All interjections and discourse acts map to concept sets, which come thru as the user input instead of what they wrote. For example *yes* and *sure* and *of course* are all treated as meaning the discourse act of agreement in the substitutes file. So you don’t see *yes, I will go* coming out of the engine. The substitutes file will remap that to the sentene *~yes*, breaking off that into its own sentence, followed by *I will go* as a new sentence.

These generic interjections (which are open to author control via substitutes.txt) are:

~yes,~no,~emogoodbye,~emohowzit,~emolaugh,~emomisunderstand
~emosad,~emohappy,~emosurprise,~emobored, ~emopain,
~emohello,~emoapology,~emoskeptic,~emocurse,~emoignorance,
~emothanks,~emodisgust,~emoprotest,~emobeg

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

The following concepts are triggered by exactly repeating either the chatbot or oneself (to a repeat count of how often repeated). Repeats are within a recency window of about 20.

```
"~repeatme","~repeatinput1","~repeatinput2","~repeatinput3","~repeatinput4","~repe
atinput5","~repeatinput6",
```

Canonization

The system actually assists you in generalizing your patterns. It simultaneously matches both the original word and a canonical form of it **if your pattern word is in the canonical form**. And it checks both lowercase and uppercase forms of your words.

For nouns, the canonical form is the singular. So if your pattern is:

?: (dog) I have a cat

this will respond equally to *I like dogs* and *I have a dog*. Whereas the pattern

?: (dogs) I have a cat

will only respond to *I like dogs* but not to *I have a dog*.

For verbs, the canonical form is the infinitive tense. If your pattern is:

*?: (be *1 correct) Yes.*

This will respond equally to *Was it correct?* and *Are you correct?* and *Is she correct?*.

Possessive suffixes *'* and *'s* transform to the word *'s*. Adjectives and adverbs revert to their base form. Determiners *a an the some these those that* become *a.* Text numbers like *two thousand and twenty one* transcribe into digit format and floating point numbers migrate to integers if they match value exactly. Personal pronouns like *me*, *my*, *myself*, *mine* move to the subject form *I*, while *whom, whomever, whoever* shift to *who* and *anyone somebody anybody* become *someone*. The file canonical.txt in LIVEDATA controls lots of these.

If the system sees & in the input, it changes it to *and.* It also changes ` to ' .

ChatScript's simple concept below accepts all tenses and conjugations of the listed verbs:

concept: ~be [be seem sound look]

If you put an apostrophe in front of a word or use words not in canonical form, the system will restrict itself to what you used in the pattern:

u: (I 'like you)   This matches I like you but not I liked you.
s: (I was)   This matches I was and Me was but not I am

Not !

The absence of words is represented using ! and means it must not be found anywhere after the current match location. When placed at the start of the pattern, it means not anywhere in the sentence at all :

*u: (![not never rarely] I * ~ingest * ~meat) You eat meat.*
*u: (!~negativeWords I * ~like * ~meat)   You like meat.*

Optional Words { }

Sometimes you can expect a word might or might not be supplied. Your pattern can reflect this, swallowed it when present. {} is just like choice [], except the match is optional. It is allowed to fail.

?: (how hot is ~number {degree deg} Farenheit) Sounds hot.
*s: (define {the word (the meaning of) } *1 >) Sorry. I don't know it.*

Note how we didn't have to say *degrees* in the optional list, because that automatically is handled by using the canonical *degree*.

Word Commands

You can issue commands to the system (prefix is colon) to inquire about words and their relationship to themselves and concepts. All commands are invisible to normal chat in that they do no affect the user's state of processing chat (except :real) , though some commands can change engine variables, which affect how the engine may process future input.

:word word – dumps the dictionary and fact and concept information about the word.

It displays everything the system knows about the given word- its parts of speech, attributes like it is a singular noun, what dictionary meanings it has, and what sets and facts it participates in directly. Just type in something like **:word tennis**

:up word - While this is interesting, for the purpose of matching, the **:up** command is more useful, because it tells you how this word participates in sets all the way up the inheritance hierarchy both of concepts and of Wordnet, so any set listed by this would be recognized if the word given as argument is used.

Suppose you are creating the concept of ~buildings. Just think of a word you want to include, like *temple*, and then use *:up temple* to see what it does.

```
For temple:
 Set hierarchy:
 Wordnet hierarchy:
 temple~1:N    means tabernacle~1 the place of worship for a Jewish congregation
   is house_of_worship~1 any building where congregations gather for prayer
     is building~3
       is construction~4
         is artefact~1
           is whole~1
             is physical_object~1
               is physical_entity~1
                 is entity~1
 temple~2:N   means temple~2 place of worship consisting of an edifice for worship of a deity
   is house_of_worship~1 any building where congregations gather for prayer
     is building~3
       is construction~4
         is artefact~1
           is whole~1
             is physical_object~1
               is physical_entity~1
                 is entity~1
 temple~3:N    means temple~3 an edifice devoted to special or exalted purposes
   is building~3 a structure that has a roof and walls and stands permanently in one place
     is construction~4
       is artefact~1
         is whole~1
           is physical_object~1
             is physical_entity~1
               is entity~1
 temple~4:N    means temple~4 the flat area on either side of the forehead
   is lineament~1 the characteristic parts of a  face: eyes and nose and mouth and chin
     is body_part~1
       is piece~11
         is thing~1
           is physical_entity~1
             is entity~1
```

If we are trying to build a concept of buildings, then temple~1, temple~2, and temple~3 are definitions that make sense (:N just names the part of speech). But notice on the up path that those definitions all come from building~3, and that using that would make sense and encompass everything that Wordnet considered a building in that sense.

:down word limit – takes a word and chases down its hierarchy showing what inherits from it. Limit is how many levels down to go (default is 2) since going down can expand into a lot of choices for some words. If the word is a concept or topic name, it displays its top level members.

SIMPLE OUTPUT

The goal of the engine is to generate output to display to a user. When a rule does that, it has accomplished the goal of the topic.

Direct Output

To generate simple output, just put the text you want to display after the pattern component of a rule. Gambits do not have to have a pattern component, in which case their output starts immediately.

```
t: This is output for the user.
?: ( hello )   How are you? Do you have a life? Are you going to die soon?
```

The system automatically manages spacing separators between words and punctuation, so *I like you?* and *I like you ?* print the same on output

AutoFormat

You pass words and punctuation for display. The system automatically formats it, so it doesn't matter if your commas and periods have spaces before them, or how many blanks or tabs there are between words. The system reformats it automatically. If you actually need to control blanks…

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.*

**Literal Output **

To output characters that have reserved meaning to the engine, like [and], you need to put a backslash in front of them. In particular, to force a newline you use \n.

Randomized Output []

You can select among a range of choices by using output choices. Each choice is encased in [], and a contiguous set of them form a zone that the system will pick randomly among. Whenever bracketed items are discontiguous, you get a new random zone.

```
?: (hi) [hello.][hi][hey]  Are you going to [dance][swim][eat] anytime soon?
```

The above has two random zones, separated by fixed text. So it might output *hello. Are you going to dance anytime soon?* or *hey Are you going to eat anytime soon?*

SIMPLE DEBUGGING

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

If the system detects bugs during execution, they go into *TMP/bugs.txt*
You can erase the entire contents of theTMP directory any time you want to.

:build xxx– reads and executes *filesxxx.txt* to rebuild some of the TOPIC data. 0 is the underlying layer (files0.txt) which depends only on the dictionary. The top layer can depend on that underlying layer. Any file can be used for the top layer, meaning you can build different bots for testing easily. You can do *:build ben (filesben.txt)* , try it out, then *:build Sharon (filessharon.txt),* etc. While a build is in progress, the log file changes to be USERS/build%s_log.txt where %s is the name you gave to the build command. If you got errors, search for ** within it. If you get errors from build that claim something is already defined and you know it isn't, you can always erase the contents of the TOPIC directory (keeping the directory) and build anew.

:build will spell check patterns, issuing warnings on items it finds dubious. You can suppress this with an optional second argument *nospell*, e.g., :build 1 nospell. You can augment this to cover output as well, by saying *:build 1 output*, though you may well get a bunch of false positives because you use slang, or from variable assignments, or whatever.

:trace all - this is the most widely used debug command and enables global tracing. After entering this, you type in your chat and watch what happens (which also gets dumped into the current log file). Problem is, it's potentially a large trace. You really want to be more focused in your endeavor.

:trace none – turns off global tracing

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

:trace !~education **-** this disables current tracing for this topic and all the topics it calls when :trace all is running. Call it again to restore to normal.

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

You can also put traces in places to watch builds (you learn about these later).

You can insert a trace command in the data of a table declaration, to trace a table being built (the log file will be from the build in progress). E.g.,

```
table: ~capital (^base ^city)
_9 = join(^city , _ ^base)
^createfact(_9 member ~capital)
DATA:
:trace all
Utah Salt_lake_city
:trace  none
```

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

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

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

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

:variables – show the values of all user variables.

:prepare sentence - This shows you how the system will tokenize a sentence and what concepts it matches. It's what it normally does to prepare to match topics against your input. You can supply *:prepare true* to make to the system do this on future sentences and then *:prepare none* to turn this off.

:postprocess sentence - This shows you how the system will tokenize your sentences as though they were output from the chatbot and runs the postprocess.

:revert – The stand-alone engine maintains a one-volley back copy of user files in */TMP*. If the chatbot answers badly, you can execute :commands like :trace all to see what is happening, then enter :revert. This automatically types the same input you did a moment ago and the chatbot should make the same mistake, this time under observation. You can keep reverting and trying again (not that it will suddenly get it right).

:testpattern (…..) sentence – the system inputs the sentence and tests the pattern against it. It tells you matched or failed.

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

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

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

:testoutput stream – the system executes the stream as though it were output and tells you what it would say.

:testtopic *topic sentence* – The runs the given topic on the sentence, with tracing automatically on. If topic is omitted, it runs the last provided topic with the sentence.

:real :command …. – prefixed before any other command, it means whatever changes to user state the command makes (like setting variables with :testoutput) will be saved into the user's state.

:repeat value – value is 0 or non-zero. It sets a global flag that lets the system repeat output in succession.

:stats - the system will output the cpu time per volley and the number of rules tested and matched.

:serverlog value - enables (1) or disables (0) logging server data into serverLog.txt

:used topicname – shows which rules have been disabled.

Testing a chatbot

The typical thing I do is sit and chat with the bot until it says something I know is wrong.

And I do the following when I see the bad input. Knowing what rule I expected to win, I trace the topic in which it resides, to see why it didn't win.

:trace ~topicname   *# turn on tracing (except for functions made quiet)*
:revert   *# go back to before I said my last thing*
read the log file or the console and see where it went wrong and why

If that isn't helpful, I may then *:trace all* and then *:revert* again.

VARIABLES

A chatbot with no ability to remember, even in the brief moment of attending to user input, would be an impoverished being indeed. ChatScript supports several levels of memorization.

_Match Variables

When you use wildcards and sets in a pattern, you can ask the system to memorize briefly the word it matches. Just place an underscore in front of what you want memorized. The purpose of memorizing is to be able to use the value on output. The results of memorization are stored on match variables named _0, _1, etc, depending upon how many underscores you use in the pattern.

    ?: (do you eat _~meat) No, I hate _0.

If the input is *do you eat ham* the output would be *No, I hate ham.* Of course, the value of _0 is only guaranteed for the execution of this rule. Match variables may be clobbered when you execute another rule. Or they may last for a while. At most it will last for the duration of the current volley (several sentences maybe) after which it should be presumed trashed.

When the system memorizes your underscore match, it stores both the original word and its canonical form. On output, by default you get the canonical form. If you want the original form, you must precede your reference with an apostrophe.

    ?: (do you eat _ [ham eggs bacon]) I eat '_0.

If the input is *do you eat eggs* the output is *I eat eggs.* Had you not used the apostrophe, the output would have been *I eat egg.*

Rarely would you ever want the canonical form of memorizing an indefinite gap.

    ?: (do you like _ or _*) I don't like '_0 so I guess that means I prefer '_1.*

If the input is *do you like eating green eggs or swimming on the beach*, the output would be *I don't like eating green eggs so I guess that means I prefer swimming on the beach.*

You are allowed _0 through _9.

If you memorize an optional area, _{test me}, then you get either the word that matched or the match variable is set to null if it fails to match.

If you use match variables within a nested level, they are discarded. E.g.,

    s: (_~fruit (_~animal) _~like)

In the above, _0 is a fruit and _1 is a like, and the _~animal is discarded when the ( ) completes.

$User_Variables

If you need memory that lasts beyond the current input, one source of this is user variables. A variable is named with a starting dollar sign and then alphanumeric letters and possibley underscores and periods. You initialize it using a C-style assignment in the output.

```
        s: ( My name is _*1 _*1 >) $firstname =  '_0  $lastname = '_1 Nice to meet you.
```

You are advised to put these computational scripts on separate lines to make it easier to read your script, but ChatScript doesn't really care.

```
        s: ( My name is _*1 _*1 >)
                $firstname =  '_0
                $lastname = '_1
                Nice to meet you.
```

The variables will last forever or until you change them. If you want the variable to disappear at the end of the volley instead, name it with $$ at the start, e.g. $$myvar.

Variable assignments extend across arithmetic operations but you cannot use parens to control operator precendence. E.g.,

```
        $myvalue = $foo + 20 * 5 / 59  This is normal output after the assignment.
```

Of course it would have been clearer to write this as:

```
        $myvalue = $foo + 20 * 5 / 59
        This is normal output after the assignment.
```

You can test variables in patterns in a variety of ways. Some such tests do not affect the current position pointer of the match. Merely putting in the variable name will ask "does it have a value"?

```
        ?: ( what is my name $firstname )  Your name is $firstname.
```

Like match variables, you can use user variables in the output. Here, if the input is *what is my name* and *$firstname* has been previously assigned to, then the pattern matches. Otherwise it fails. You are free to refer to variables that don't exist. The pattern will simply fail.

You can also directly test a variable to see if it has a particular value using a relational test in the pattern. One such relation is =.

```
        ?: ($gender=male I like boys)  Oh, dear.
```

Here, the rule will only start checking for input matches if *$gender* has the value *male*. If it is not defined or has any other value, this rule fails immediately. A relational test requires the two sides of the relation and the relation symbol all be jammed together with no spaces. So the following rule is tantamount to seeing if *$gender* has ever been assigned to, followed by seeing if the user typed *=male* anywhere.

```
        ?: ($gender   =male )
```

Other relations are < and >, which will require the system convert the variable's text value into numeric.

```
        ?: (I am _~number years old _0<10) You are a child.
```

You can invert a relation test using the ! operator.

```
        ?: (I am ~number years old !_0<10) You are not a child.
```

You can assign match variables manually, though unless you are assigning from a match variable, you get no canonical data.

s: (my life) _8 = hello
*s: (my _*1)   _8 = _0*

In the first example, _8 gets the word hello for both canonical and original forms. It doesn't process it. In the second example, since _0 has dual form, the assignment is dual form.

Another relation is the *?* which means set membership. It asks if the item on the left is a ultimately a member of the set on the right. In this contrived example, one imagines:

concept: ~pork [bacon ham]
concept: ~meat [~pork ~beef ~chicken]
…topic definition
?: (I have _~meat _0?~pork)

The example is contrived because one would in this case merely write:

?: (I have _~pork)

and not bother with ~meat, but one can imagine cases where you are trying to intersect with an unrelated set. For example:

concept: ~winplaceshow [first second third 1st 2nd 3rd]
…topic definition
?: (I came in _~placenumber _0?~winplaceshow) Congratulations.

You can ask if the contents of the variable are a word in the sentence by putting an empty set membership relation. Since there is no set, the only context is has is the sentence itself as a set. This does change the match position if it is found.

?: (is your name $firstname?) Yes

System-used $variables

$bot – the id of the current bot in control. Set by system when a new user starts up.
$login - set by the system to the current login id. May be user entered login or concatenated with ip when running from open internet.
$loebner- defined if this is a loebner competition engine
$randindex- the current random seed for this volley – set by engine
$control_pre – name of topic to run in gambit mode on pre-pass, set by author
$control_main – name of topic to run in responder mode on main volleys, set by author
$control_post – name of topic to run in gambit mode on post-pass, set by author
$code – which of *pre*, *main*, *post* is running at present – set by engine
$token- bits controlling how the tokenizer works. By default when null, you get all bits assumed on.

1: DO_SUBSTITUTES – use the substitutes.txt file to adjust words
2: DO_NUMBER_MERGE – merge text numbers into single tokens
4: DO_PROPERNAME_MERGE – merge proper names into single tokens
8: DO_SPELLCHECK – adjust spelling of badly spelled words

%System Variables

The system has some predefined variables which you can generally test and use but not normally assign to. These all begin with % .

%rand – get a random number

%date – one or two digit day of the month
%day - Sunday, etc
%daynumber – 0-6 where 0 = sunday
%second – 0-59
%minute – 0-59
%hour – 0-23
%time - hh:mm:ss
%week - 1-5 (week of the month)
%month - 1-12 (January = 1)
%monthname – January, etc
%year – e.g., 2011

%input – the count of the number of inputs this user has made total over all
%length – the length in tokens of the current sentence
%tense – past , present, or future
%topic – name of the current "real" topic . if control is currently in topic which is not *system*,*disabled*, or *nostay*, then that is the topic. Otherwise the most interesting topic is found.
%userfirstline – value of %input that is at the start of this conversation start
%response – number of responses that have been generated for this sentence
%lastoutput – the text of the last generated response

Values below are all true /false and will be "1" or null.

%more – is there another sentence after this
%morequestion – is there a ? in the remaining sentence inputs
%lastquestion – was the last generated response a question (has a ?)
%question – was the user input a question – same as ? in a pattern
%regression – is the system running a regression test
%server – is the system running in server mode
%rejoinder- does the system have a rejoinder for its output

You actually can assign to any of them. This will override them and make them return what you tell them to and is a particularly BAD thing to do if this is running on a server since it affects all users. Typically one does this one in a #! comment line to set up conditions for testing using *:verify.*

Long-term variables
The system normally stores variables on a per-user basis. You can set bot-specific facts in the login function of a bot. If you have facts you want to be global across all bots and as

part of the base system, you can put those assignments into a table, read in under a :build command.

Facts

The ultimate variable is the fact, but that is way too advanced for now.

ADVANCED TOPICS

There are several things to know about advanced topics.

Topic Control Flags

The first are topic flags, that control its behavior. These are placed between the topic name and the (keywords list). You may have multiple flags. E.g.

topic: ~rust noerase random [rust iron oxide]

The flags and their meanings are:

Random - search rules randomly instead of linearly
NoRandom – (default) search rules linearly

NoErase – do not mark responders as used up when they generate output. Gambits are always erased and rejoinders never erased (but the top level controlling it may be).
Erase – (default) erase rules that successfully generate output

NoStay – do not consider this a topic to remain in, leave it (except for rejoinders).
Stay – (default) make this an interesting 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, NoErase. You can change the NoStay property, but not the NoErase property.
It never participates in values from ^gambittopics(), even if it has gambits.
System topics can never be considered interesting (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.

Bot=name – if this is given, only named bots are allowed to use this topic. You can name multiple bots separated by commas. E.g.,

topic: ~mytopic bot=harry,Georgia,roman [mykeyword]

Interesting Topics

The second thing to know is what makes a topic *interesting.* Control flow passes through various topics, some of which become interesting, meaning one wants to continue in those topics when talking to the user. Topics that are considered always uninteresting are:

system topics, disabled topics (you can disable a topic so it won't execute), and nostay topics.

What makes a remaining topic interesting 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 interesting. 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 interesting topics. The order is: 1st- being within that topic executing rules now, 2nd- the most recently added topic (or revived topic) is the most interesting.. You can get the name of the current most interesting topic (*%topic*), add interesting topics yourself (*^addtopic*()), and pop the most interesting 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.

Variable Remaps

Match-variable names like _1 are fine when you are referring to a particular match from a pattern. But you can use match variables as transient global variables and such names are not mnemonic. You can “rename” match variables as follows:
define: _monthname _5   -- whenver the system sees _monthname, it will use _5
You can do this within a topic (limited to it) or outside a topic (globally thereafter).

ADVANCED PATTERNS

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.

^eval – to evaluate a series as though it were output (like to assign a variable)

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

Macros

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

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

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

The above patternmacro takes one argument (who we are talking about). After checking that the sentence is not in the negative, it uses a choice to consider alternative ways of asking what the hair color is. The first way matches *are you a redhead*. The second way matches *what is my hair color*. The call passes in the value *I* (which will also match *my mine* etc in the canonical form). Every place in the macro code where ^who exists, the actual value passed through will be used. ??? what about ( ) as an argument or []/?/??

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
```

Backward Wildcards *-1

Not only can you match forwards, you can actually request a particular word n words earlier than your current position. This is sometimes useful when you find the main keyword like a noun, and then want to go back and find out any adjective modifying it.

?: (_~pet _-1 _1?~adjective) A '0 '1?*

This pattern reacts to *Do you have a red dog* by matching the essential word dog (as ~pet) and then backing up and grabbing the preceding word and testing to see if it is an adjective. If so, the response is *A red dog?*

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 quote 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

**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?*

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)

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 make pass patterns data 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 parents), or a macro argument. The contents of the macro argument are used in the pattern in its place. But macro arguments only exist inside of macros. But macros don't let you write rules, only pieces of rules.

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.

The other thing you can do is change pattern context by position a pattern to a prior match location. 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 )
        $result = null
        respond(~GATHERNOUNPHRASE)
        if ($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
```

ADVANCED OUTPUT

Simple output puts words into the output stream. When the rule successfully completes, that stream is stored away to be said to the user, unless the output was destined for storing on a variable or becoming the argument to a function or macro.

What I didn't tell you before was that if the rule fails along the way, an incomplete stream is cancelled and says nothing to the user. 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.

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. 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 nothing because after starting output, the loop fails and cancels the output.

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 return a fail code it passes
4. A relation – one of == != < <= > >=

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
```

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.
Wheras 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)
```

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
```

Control Functions

^**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** or it's a fail(rule) in the body of a **loop**, it merely stops those and not anything broader. The failure codes are:

- **RULE** –stops the current rule
- **TOPIC**- stops not only the current rule also the current topic.
- **SENTENCE** – stops the current rule, the current topic, and the current sentence.

If you issue a fail(topic), then rule processing stops for that topic, but as it exits, it passes up to the caller a downgraded fail(rule), so the caller can just continue executing other rules.

^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.

^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** or is end(rule) in the body of a **loop**, it merely stops those. 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.

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

^system(any number of arguments) - the arguments, separated by spaces, are passed as a text string to the operating system for execution as a command. The function returns RULE fail if the operating system returns a non-zero value. You can transfer data back and forth via files by using ^import and ^export of facts.

^enable (what name) – what can be *topic* or *rule.* If what is topic, then name must be a topic name. It enables the topic if it was disabled (by default topics are enabled). If what is a rule, then name is a rule label in the current topic or is a dotted pair of topicname and label. If what is *disable*, it alters whether the system can disable a new matching rule or not (any non-0 value means it can).

^disable (what name) – what can be *topic* or *rule* or *rejoinder.* If what is topic, then name must be a topic name. It disables the topic, which means the system will automatically fail execution. If what is a rule, then name is a rule label in the current topic or is a dotted pair of topicname and label. It means the rule is marked as used up, so it cannot be used. If what is a *rejoinder* then it will cancel any pending rejoinder (no name argument is needed).

^retry() - rexecute the current rule. You better change SOMETHING in the world, or it will just infinitely loop on itself.

^input(…..) - the arguments, separated by spaces, are feed back into the input stream as the next 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.

^mark (word match_variable match_variable) – the word or topic or concept name is marked as though it had been in the input at the point 1st match variable. If that match variable is omitted, then at the last element of the pattern matched. If the second match_variable is supplied, this marks the end of a phrase being matched, otherwise it defaults to the first match_variable (a 1-word phrase).

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*.

^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. If match-variable is omitted, the last word of the sentence is used.

^analyze (stream) - the contents of the stream (one sentence) are analyzed like input to the system, so that thereafter patterns and rules operate on it. Used to analyze the chatbot's output – things like auto-pronoun handling, etc.

^setquestion (value) – value is 0 or 1 and it sets the internal flag that indicates that the current sentence is a question or not.

^reset (what ?) – what can be *user* or *topic*. If user, the system drops all history and starts the user afresh from first meeting. If topic, you must name the topic. It will re-enable all rules within the topic. If you name no topic, ALL topics are reset.

^refine() - this is like a switch statement in C. It executes in order the rejoinders attached to its rule. When the pattern of one matches, it executes its output and is done, regardless of whether or not the output fails.

^blacklist(minutes) – do not respond to this user ip for the next minutes count.

^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.

Topic Functions

^rejoinder( ) – see if the prior input ended with a potential rejoinder rule, and if so test it on the current sentence.

^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.

^gambit(topic-name) - runs the named 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. The topic-name may be ~, which means use the current topic you are within. If the topic-name is omitted entirely, the system will walk the interesting 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 interesting topics for the future.

^gambittopics() – finds user topics (not system topics) with gambits remaining. If you use it in a fact-set assignment statement, it stores all topics found as facts *(topicname ^gambittopics ^gambittopics).* You can then display them or use them as you wish E.g.

```
        @1 = ^gambittopics()
        ^gambit( ^pick(@1))            #  randomly issue a gambit
```

Otherwise, if you don't use an assignment, it outputs the first topic found into the output stream and stops. If no topics are found, the function returns RULE fail.

^keywordtopics() – gets the most salient topics covering the input (excluding system, or disabled topics). Usually stored into a fact set as facts, if not within an assignment just the most salient is put into the output stream. Facts are most salient first. E.g.

```
        @7 = keywordtopics()
        ^gambit(first(@7subject))
```

^addtopic(topicname) – adds the named topic as an interesting topic. 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 it to the interesting list. See %topic and ^popic() for accessing this list.

^poptopic(topicname) – removes the named topic as an interesting topic. The intent is not to automatically return here in future conversation. If topicname is omitted, removes the current topic.

^reuse(rule-label optional-enable) - 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. ^reuse stops at the first correctly labeled rule it can find and issues a RULE fail if it cannot find one. 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.

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 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)
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

^find(setname itemname) – given a concept set, find the ordered position of the 2nd argument within it. Output that index. Used, for example, to compare two poker hands.

Output Functions

^preprint (xxxx) - the stream of xxxx 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. This is used, for example, to do transitional phrase printing. Once the system has generated output, you can know what topic it came from and what topic it started out in. If it has changed topics, you can insert a transitional sentence at the front. A rule that performs preprint does not automatically try to erase itself. Only if data is left in the output stream when a rule ends will that data be output normally and the rule try to erase itself.

Word Functions

^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.

NOUN x SINGULAR, PLURAL, PROPER, 1, other positive numbers

This converts the word to the appropriate designation.

VERB x PARTICIPLE, INFINITIVE, PAST

DETERMINER x –

This outputs the determined 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.

AUX x pronoun – generate correct form of auxiliary verb x, given personal pronoun I,you,he/she/it.

^define (word) – output the definition of the word

^length(word) – puts the length of the word into the output stream. If word is actually a fact set reference (e.g., @2), it returns the count of facts in the set.

^join (any number of arguments) – concatenates them all together, putting the result into the output stream.

^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.

^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*

^spellmatch(word pattern) – given a text string and a pattern (as defined by ^spell), see if the word matches the pattern.

^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*

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

^nolowercase(word) – finds if a word has a lower-case form in the dictionary, failing if it does. 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.

^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

^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 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.

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

Concept Functions

^pick (~concept) – retrieve a random member of the concept. Pick is also used with factsets to pick a random fact (see FACTS).

Debugging Functions

^log (any number of arguments) – dumps the argument values into the log file.

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*

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]
```

The # character revisited

Yes, normally # means comment. But it is also used to access predefined number constants from ChatScript Engine. I said earlier that you could use a #define constant from dictionarySystem.h by using #name in the comparison syntax like

```
u: ( _0=#NOUN )
```

So it should come as no surprise that you can actually do that as a stand-alone item in any pattern (for which there is probably no use) and in output. It is useful, for example in setting the $token value to control tokenization. E.g.,

```
u: () $token = 255 - #DO_SPELLCHECK   # removes spell check use
```

ADVANCED DEBUGGING

Authorization to use the **:** commands comes from reading the file **authorizedIP.txt** which names *all* or is a list of acceptable IP addresses (so no user can command a chatscript server inappropriately.)

: facts – this writes all current facts into a file *TMP/facts.txt*. It allows you to see everything the system knows at the moment.

:debug – used to get breakpoint control over the system when debugging the engine. You add this to some data source file prior to building, just before you want to debug, and set the breakpoint in the :debug code.

:shutdown- exit the program (used to restart a server version completely)

:bot – botname – in non-server mode, when you have multiple bots compiled together, you have to remember to append a botname to your login or you get the default bot. This allows you to change after you have forgotten, and will start a conversation with the named bot instead.

:verifysub - runs a regression test to prove the substitutes in LIVEDATA/substitutes.txt are all working correctly.

:execute function(args) – execute the function and args specified. Technically it executes the output data given, but there is little point in it being other than a function call.

:verify ~topicname – for the named topic (or all topics if none is given) it looks for rules that have special verification data associated with them that allows it to test the pattern of the rule to see if it is correctly handled. Before any rule you can put a comment in the form:

```
#!  This is sample input
?: (sample)  this is my rule
```

The system will use the comment as input and see if the rule can match it. It does not execute the output. Rules that need variables set certain ways can do variable assigns at the end of the comment. You can also have more than one verification line before a rule.

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

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

Another form is !x, which says this is just a comment to be displayed by the :abstract command.

If :trace has been set non-zero, then tracing will be turned off during verification, but any rules that fail will be immediately be rerun with tracing on.

This testing is for patterns, given that you actually get to the rule itself. But you might not get to the rule if some rule earlier in the topic has a matching pattern. But you can test for that too, though the report of failures may be extensive and tedious.

:verify topic and *:verify topic ~topicname* will test the entire topic and report on rules that are blocked by earlier rules. This presumes you are within the topic and says nothing about whether the rule could be reached if you were outside the topic (i.e., can keywords of the input get you to that topic in the first place).

Blocking may mean you should reorder the rules. Or it may happen because you have multiple rules to cover the same input (after one gets erased). Or maybe you just redundantly covered things and should eliminate a rule. If you know the rule will fail and don't want it reported, mark its comment as *#!!T* instead of just *#!*. This will allow it to be tested with rule testing, but not topic testing. By the way, the corresponding full form of the original command is *:verify pattern* and *:verify pattern ~topicname* but the system detects if you omit the kind *pattern* and defaults to it.

And, a rule may not be findable if outside the topic when the input lacks appropriate keywords. You can test for this with *:verify outside* and *:verify outside ~topicname*. This reports sample inputs that contain no keywords of the topic. This may or may not be something you should alter. Depends upon your purpose.

:abstract – prints out an abstract view of the entire topic data or the named topic. This shows the structure (gambits, rejoinders, responders) as well as conditions on gambits and normal text content of everything, but omits actual code complexity. It is useful for seeing what will be said in response to sample input (if you use the #! and #!x commands on rules. E.g.

```
****** Topic: ~introductions[]
t: ( $old   %input=0   %hour<12 $name ) Good morning, .
t: ( $old   %input=0   %hour>11   %hour<18 $name ) Good afternoon, .
t: ( $old   %input=0   %hour>18 $name ) Good evening, .
t: Where do you live?
   a: "Fukushima" =>   I've heard of _0. Were you born there?
   a: "I live in Japan" =>   I've visited Japan.
   a: "I live in California" =>   That's where I live!
   a: "Libya" =>   I would have thought you lived in Japan, not _0 .
   a: "Earth" =>   Yes, we all live on Earth.
   a: "Mars" =>   I don't believe you.
t: What do you do for a living?
u: "Am I welcome here?" =>   Of course you are welcome.
s: "I'm back" =>
  [ Where did you go?  ]
  [ Where have you been?  ]
  [ I'm glad.  ]
s: "Knock" =>   Who's there?
```

While :abstract primary does topics, you can get it to do fact data as well. It will attempt to call a function ^abstract_facts(), so if you define that, you can do whatever you want for abstracting facts.

:fakereply value – instead of getting input from you, it always makes the input *OK* if value is 1 or *WHY?* if value is 2. This is a stress test for systems, to prove they keep going without crashing.

:source filename – switches to reading input from a file instead of stdin. The system normally prints out just the output, while the log file contains both the input and the output. You can say *:source filename echo* to have input echoed to the console. If you say *:source filename internal* the system will echo the input, then echo the tokenized sentences it handled.

:restart – makes the system reload its data files (dictionary, topics, livedata) on the fly. This is useful if you have a live server and want to upload changes into it.

:trim directory value - this reads all the log files in a directory and writes out a file *x.txt* in a format specified by value. 2 means show the users input on one line and indent the computer's output on the next. 5 means show on one line the computer's output followed by => and the the user's output. The 5 format is good for finding all the ways humans have responded to some output you make, so you can plan for clever rejoinders.

:topics – displays a list of all topics defined.

:help- displays available commands and a brief statement of purpose

:dumpset conceptname value – this dumps a concept ontology in various formats.
:dumpset ~nouns 0 shows the noun hierarchy tree by concept (without showing the actual words). For each concept, it shows how many words it covers.
:dumpset ~nouns 1 shows the same tree, but populates it with the actual vocabulary covered.

:attopic – for all topics this dumps both the sample input from the verification comment (if there is one) and the normal text (w/o fancy script code) of what the system would say. It's a more legible non-technical overview of your topic code.

SIMPLE FACTS

Facts are triples of subject, verb, object – though those are just the names of the fields and may or may not be their content. Facts look like this:

(Bob eat fish)

The system has a number of facts it comes bundled with and others can be created and stored either from compiling scripts, or from interactions with the user. Facts can use words, numbers, or other facts as field values, representing anything. You can build records, arbitrary graphs, treat them as arrays of data, etc.

Simple Creating Facts

^createfact(subject verb object) – this creates a fact triple. The system will not create duplicate facts. If you have a fact *(Bob eat fish)* then executing

^createfact(Bob eat fish)

will do nothing further (but it will return the found fact). One way to create a fact of a fact is as follows:

^createfact((Bob eat fish) how slowly)

The other way is to assign the value of fact creation to a variable and then use that variable. You need to pass in a flag at creation, to tell the system the value is a factid.

$fact = ^createfact(Bob own fish)
^createfact ($fact Bob pet FACTSUBJECT)
$fact = ^createfact(Bob own dog)
^createfact ($fact Bob pet FACTSUBJECT)

The above creates facts which are findable by querying for pets Bob has. You can have any number of flags at the end. Other flags include:

FACTVERB and FACTOBJECT
TRANSIENT – the fact will disappear at the end of this volley

Accessing Facts

To find facts, you need to make a query. There can be many different kinds of queries.

^query(kind subject verb object) – The simplest query names the kind of query and gives some or all of the field values that you want to find. Any field value can be replaced with ? which means either you don't care or you don't know and want to find it. The kinds of queries are programmable and are defined in LIVEDATA/queries.txt (but you need to be really advanced to add to it). The simplest query kinds are:

direct_s - find all facts with the given subject
direct_v – find all facts with the given verb
direct_o – find all facts with the given object
direct_sv – find all facts with the given subject and verb
direct_vo – find all facts with the given object and verb
direct_svo- find all facts given all fields (prove that this fact exists).

Unipropogate – find how subject joins into the object set.

If no matching facts are found, the function returns the RULE fail code.

?: (do you have a dog) ^query(direct_svo I own dog) Yes.

If the above query finds a fact *(I own dog)* then the rule says *yes*. If not, the rule fails during output. This query could have been put inside the pattern instead.

For unipropogate, if you have these concepts;
concept: ~things (~animals ~vegetables ~minerals)
concept: ~animals (~canine ~feline)
concept: ~canine (dog)

Then ^query(unipropogate dog ? ~things 1) would return (~animals member ~things). Note that the set to be found (~things) is not expanded. Normal queries expand any reference to a set into all of its members. But unipropogate expects a set as its object argument, so it does not need to be quoted.

System-reserved verbs

The system builds the Word-net hierarchy using the verb *is*, with the lower-level (more specific) word as subject and the upper-level word as object. E.g.,
(dog~1 is animal~4)
The system builds concept and topic sets using the verb *member* with the member value as subject and the set name as object. E.g.,
(run member ~movementverbs)
When you build a table and a data member has a short-form like *Paris* for *Paris,_France*, the verb is also *member* with subject as short form and long form as object. E.g.,
(Paris member Paris,_France)

@Fact-Sets

The results of queries are stored in a fact-set. Fact-sets are labeled @0, @1, etc. through @9. By default in the simplest queries, the system will find all facts that match and store them in fact-set @0. A fact set is a collection of facts, but since facts have fields (are like records), it is also valid to say a factset is a collection of subjects, or verbs, or objects. Therefore when you use a factset, you normally have to specify how you want it used.
@1subject means use the subject field
@1verb means use the verb field
@1object means use the object field
@1fact means keep the fact intact (a reference to the fact) – required if assigning to another set.
@1+ means spread the subject,verb,object onto successive match variables – only valid with match variables

@1- means spread the object,verb,subject onto successive match variables– only valid with match variables
@1all means spread subject,verb,object,flags onto successive match variables. – only valid with match variables
_6 = ^first(@1+)   - this puts subject in _6, verb in _7, object in _8

?: (do you have a pet ^query(direct_sv I pet ?)) I have a @0object.
If the chatbot has facts about what pets it has stored like *(I pet dog)* and *(I pet cat)*, then the rule can find them and display one of them. Which one it shows is arbitrary, it will be the first fact found.

You can transfer the contents of one fact-set to another with a simple assignment statement like @2 = @1 .

You can transfer fields of a fact from a fact-set using assignment, while simultaneously removing that fact from the set. The functions to do this are:
^first(fact-set) – retrieve the first fact
^last (fact-set) – retrieve the last fact
^pick (fact-set) – retrieve a random fact

You can erase the contents of a fact-set merely by assigning null into it.
@1 = null
This does not destroy the facts; merely the collection of them.

You can sort a fact set which has number values as a field.
^sort(fact-set) – the fact set is sorted from highest first. By default, the subject is treated as a float for sorting. You can say something like @2object to sort on the object field.

If you actually want to destroy facts, you can query them into a fact-set and then do this:
^delete(@1) – all facts in @1 will be deleted and the set erased

If you want to know how many facts a fact-set has, you can do this:
^length(@1) - outputs the count of facts

A special count function exists to tell you how many members a concept set has.
^countmember(concept-topic-name) – this will tell you how many members are in this concept or topic at the top level (not recursive).

^Unpackfactref examines facts in a set and generates all fact references from it. That is, it lists all the fields that are themselves facts.
@1 = ^unpackfactref(@2)
All facts which are field values in @2 goto @1.

Unlike variables, which by default are saved across inputs, fact sets are by default discarded across inputs. You can force a set to be saved by saying:
^save(@9 true)   # force set to save thereafter

```
^save(@9 false)                    # turn off saving thereafter
```

Fact Indexing

A fact like *(bird eat worm)* is indexed by the system so that *bird* can find facts with bird as the subject or as the verb or as the object. Similarly *eat* can find facts involving it in each position. As a new fact is added, like *(bird hate cat)* the word *bird* gets the new fact added to the front of its list of facts involving bird in the subject field. So if you search for just one fact where bird is the subject, you get the most recent fact. If you search for all facts with bird as the subject, the facts will be stored in a fact set most recent first (lowest/earliest element of the fact set). You would use ^first(@2) to get its most recent fact and ^last(@2) to get its oldest fact.

Tables

With the ability to create and manipulate facts comes the need to create large numbers of them conveniently. This is the top-level declaration of a table, a combination of a transient output macro declaration and a bunch of data to execute the macro on. Usually the macro creates facts.

The table has a name (ignored- just for your documentation convenience), a list of arguments, a bunch of script, a DATA: separator, and then the table data. The data is line-oriented. Within a line there are no rules about whitespace; you can indent, tab, use lots of spaces, etc. Each line should have as many elements as the table has arguments. The table ends with the end of file or a new top-level declaration. E.g.,

```
Table: authors (^author ^work ^copyright)
        ^createfact(^author member ~author)          # add to concept ~author
        ^createfact(^work member ~book)              # add to concept ~book
        ^createfact(^work exemplar ^author)          # author wrote this book
        if (^copyright != *) { ^createfact(^copyright date ^work) }
Data:
“Mark Twain”  “Huckleberry Finn” 1884
“Mark Twain” “Tom Sawyer” *       # don’t know the date
```

For tables with really short data, you can choose to cheat on the separate line concept, and separate your entries with \n , which is the equivalent.

```
DATA:
a 1 \n b 2 \n c 3 \n d 4 \n e 5   # values assigned to letters.
f 6 g 7
```

A table allows you to automatically list shortened synonyms of proper names. For example, *Paris* is a shortened synonym for *Paris, France.* In a table of capitals, you would normally make the fact on the full name, and write the shortened synonyms in parens. You may have more than one:

```
“Paris, France” (Paris “City of Love”)  France
```

These synonyms are represented using the *member* verb, sort of like making a concept set of the full name. The system detects this specially during inferencing, and if an argument to ^query were *Paris*, it could automatically transfer across and consider facts for *Paris,_France* as well. It would not go the other way, however, so if the argument were *Paris_France*, it would not move over to *Paris.* You should store your facts on the full name. The mechanism allows user input to use the short name.

While a line of table data must fill all fields of the table exactly (no more or less), you can tell the system to fill in the remaining arguments with "*" by putting "…" as your last value. Eg.

```
table: test(^item1 ^item2 ^item3 ^item4)
....
Data:
lion 50 ...
```

This table will use * for item3 and item4 of lion.

ADVANCED FACTS

Facts of Facts

Suppose you do something like *CreateFact(john eat (wet food peanuts))*. What happens when you retrieve it into a fact set and then do *_1 = ^last(@1)* and get the fact disassembled onto _1, _2, _3, and *4?* What you get for _3 is a reference to a fact, that is, a number. You can decode that by using *^fact(3 subject) or ^fact(_3 verb) or ^fact(_3 object)* to get *wet* or *food* or *peanuts*. The first argument to ^fact is a fact number.

You get a fact number if you do _3 = CreateFact(….) and can decode _3 the same way. Naturally this function fails if you give it something that cannot be a fact reference.

Reading and Writing Facts

You can write a file of facts with ^export(filename @setid). Filename is either a full path including drive or a path below the chatscript root directory. All non-dead, non-null facts in the set will be written to the file, one fact per line. If the file cannot be written or the setid is illegal, it returns FAILRULE. The flags of the fact will NOT include TRANSIENTFACT.

You can read a corresponding file with ^import(filename @setid erase transient). Filename has the same interpretation. The second argument names the set where the digested facts will be stored. It is not cleared so incoming facts augment it. And if a fact repeats one already existing, no new fact is created but the set will have the old fact added to it.The third argument, if the word erase, will erase the file after reading it. Any other value is ignored. The fourth argument, if transient, will mark the facts as transient and they will disappear automatically at the end of the volley (so you better process them immediately). Any other value will be ignored and the facts will enter the user's space permanently. If the file cannot be read, it returns FAILRULE.

A rule file contains one rule per line. Rule format is like DICT/facts.txt

(subject verb object)

or

(subject verb object flags)

The fields subject, verb, and object should contain no embedded blanks, unless the entire value is encased in double quotes. The fields may be normal or they may be facts themselves. E.g.
((_ ~foods My_favorite_cuisine_is_Chinese.) benfavorite cuisine 0x80)

Flags, if supplied, should be a hex number or a normal integer number, representing property bits you want enabled (see values in facts.h).

ESOTERIC FACTS

Compiled Script Table Arguments

You can specify that a table argument string is to be compiled as output script. Normally it's standard word processing like all English phrases. To compile it, you prefix the doublequoted string with the function designator ^. E.g.,

DATA:

~books "this is normal" ^"[script a][script b] ^fail(TOPIC)"

This acts like a typical string. You pass it around, store it as value of variables or as a field of a fact. Like all other strings, it remains itself whenever it is put into the output stream, EXCEPT if you pass it into the *^eval* function. Then it will actual get executed So. To use that argument effectively, you would get it out of the fact you built and store it onto some variable (like _5 or $value) , and then ^eval(_5) or ^eval($value).

FactSet Remaps

Factset names like @1 are not mnemonic. You can "rename" them as follows:
define: @authors @5 -- whenver the system sees @authors, it will use @5
You can do this within a topic (limited to it) or outside a topic (globally thereafter).

Defining your own queries

The query code wanders around facts to find those you want. But since facts can represent anything, you may need to custom tailor the query system, which itself is a mini-programming language. The full query function is takes nine arguments and any arguments at the end you omit default themselves.

All query kinds are defined in LIVEDATA/queries.txt and you can add entries to that (or revise existing ones). The essential things a query needs to be able to do is:

1. Start with existing words or facts
2. Find related words or facts
3. Mark newly found words or facts so you don't trip over them multiple times
4. Mark words or facts that you want to ignore or be treated as a successful find
5. Store found facts

A query specification provides a name for the query and specifies what operations to do with what arguments, in what order.

An essential notion is the "tag". As the system examines facts, it is not going to compare the text strings of words with some goal. That would be inefficient. Instead it looks to see if a word or a fact has a particular "tag" on it. Each word/fact can have a single tag id, drawn from a set of nine. The tags ids are labeled '1' thru '9'.

Another essential notion is the field/value. One refers to fields of facts or values of the incoming arguments, or direct values in the query script. Here are the codes involved:

1. s = refers to the subject argument or the subject field of a fact
2. v = refers to the verb argument or the verb field of a fact
3. o = refers to the object argument or the object field of a fact
4. p = refers to the propogate argument
5. m = refers to the match argument
6. ~set = use the explicitly named concept set
7. 'word = use the explicitly named word
8. @n = use the named fact set

Each query has is composed of four segments. Each segment is separated using a colon. Each segment is a series of actions, which typically involve naming a tag, a field, and then the operation, and possibly special arguments to the operation.

You can separate things in a segment with a period or an underscore, to assist in visual clarity. Those characters are ignored. I always separate actions by underscores. The period I use to mark the end of literal values (~sets and 'words).

EXAMPLE 1 – PARIS as subject

Consider this example: we want to find facts about Paris. The system has these facts:

(Paris exemplar France) and (Paris member ~capital)

Our query will be ^query(direct_s Paris ? ?) which request all facts about a subject named Paris (to be stored in the default output factset @0).

Segment one handles marking and/or storing initial values. You always start by naming the tag you want to use, then naming the field/value and the operation. The operations are:

1. t = tag the item
2. q = tag and queue the item
3. < or > scan from the item, tagging things found (more explanation shortly)

The query *direct_s,* which finds facts that have a given subject, is defined as

1sq:s::

This says segment 1 is *1sq* and segment 2 is *s* and segments 3 and 4 have no data. Segment 1 says to start with a tag of '1', use the subject argument and tag and queue it.

Segment two says how to use the queue. The queue is a list of words or facts that will be used to find facts. In our example, having stored the word *Paris* onto the queue, we now get all facts in which Paris participates as the subject (the *s:* segment *)*

Segment three tells how to disqualify facts that are found (deciding not to return them). There is no code here, so all facts found will be acceptable.

Segment four tells how to take disqualified facts as a source of further navigation around the fact space. There is nothing here either.

Therefore the system returns the two facts with Paris as the subject.

Example 2 – Finding facts up in the hierarchy

Assume you have this fact   (23 doyou ~like) and what you actually have is a specific verb *like* which is a member of ~like. You want to find facts using *doyou* and *like* and find facts where *doyou* matches and some set that contains *like* matches. The query for this is *direct_v<o*, which means you have a verb and you have an object but you want the object to match anywhere up in the hierarchy. < , which means the start of the sentence in patterns, really means the left side of something. And in the case of facts and concepts, the left side is the more specific (lower in the hierarchy) and the right side is most general (higher in the higherarchy) when the verb is member.

POS-Tagging and Pseudo Parsing

ChatScript does not have a built-in parser because it degrades performance too much. But it does have a Part-Of_Speech (POS) Tagger to try to label how words get used.

The result of this is two-fold. First, words are marked with concepts reflecting their parts of speech. And second, if you use part-of-speech limitations on keywords, like *strike~v* (meaning only the verb use of like), then the system can avoid reacting with *"Why did you hit him"* when the input is *"He went on strike."*

The pos-tagger uses dictionary information and a collection of grammar rules to decide on word use. Those rules are in LIVEDATA/posrules.txt if you want to alter them.

The concepts that might be marked are high level and refined. A word can be marked with both where appropriate, like NOUN and NOUN_SINGULAR.

The high concepts and refined concepts (indented) are:

```
~NOUN
        ~NOUN_SINGULAR          ~NOUN_PLURAL
        ~NOUN_PROPER_SINGULAR   ~NOUN_PROPER_PLURAL
        ~NOUN_GERUND  ~NOUN_CARDINAL  ~NOUN_INFINITIVE
~VERB
        ~VERB_INFINITIVE        ~VERB_PRESENT
        ~VERB_PRESENT_3PS       ~VERB_PRESENT_PARTICIPLE
        ~VERB_PAST      ~VERB_PAST_PARTICIPLE
~AUX_VERB
        ~AUX_VERB_PRESENT   ~AUX_VERB_PAST
        ~AUX_VERB_FUTURE
~ADJECTIVE
        ~ADJECTIVE_BASIC        ~ADJECTIVE_MORE
        ~ADJECTIVE_MOST         ~ADJECTIVE_PARTICIPLE
        ~ADJECTIVE_CARDINAL ~ADJECTIVE_ORDINAL
        ~ADJECTIVE_INFINITIVE
~ADVERB
        ~ADVERB_BASIC  ~ADVERB_MORE  ~ADVERB_MOST
~DETERMINER
~PREPOSITION
~CONJUNCTION_COORDINATE         ~CONJUNCTION_SUBORDINATE
~PRONOUN_SUBJECT ~PRONOUN_OBJECT~PRONOUN_POSSESSIVE
~POSSESSIVE
~TO_INFINITIVE ~THERE_EXISTENTIAL
~COMMA ~PAREN
~PARTICLE
~WH_ADVERB
```

Some of those are obvious and some aren't.

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

~number is not a part of speech, but is comprise of ~noun_cardinal (a normal number value like 17 or seventeen) and ~adjective_cardinal (also a normal numeral value) and ~adjective_ordinal (which is also ~placenumber) like *first.*

To can be many things. When used in the infinitive phrase *To go*, it is marked ~to_infinitive. ~verb_infinitive gets special handling. It refers both to a match on the infinitive form of the verb, as well as the phrase with to attached—i.e., *I like to swim* will match (I like to ~infinitive) as will (I like ~infinitive). This allows you to use a pattern like (I like [~infinitive ~participle]) treating *I like running* and *I like to run* as being the same meaning. This fails if the to is separated by an adverb as in *To boldly go*.

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

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

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

~wh_adverb refers to classic question words like *Where are you, When will we go,* etc.

~possesive refers to *'s* and *'* that indicate possession, while possessive pronouns get their own labeling *~pronoun_possessive*. ~pronoun_subject is a pronoun used as a subject (like *he)* while pronoun_object refers to objective form like (*him)*

Pseudo Parsing

You can simulate some parsing using patterns. Tommorow begins the submission month for this year's Loebner Prize. While I doubt anyone other than me can get together a ChatScript-based entry in time, I thought I would provide some code that would assist in that direction and teach esoteric features of ChatScript (not that you need esoterica yet, since you are still digesting the basics).

The ancestor of ChatScript, CHAT-L, had a built-in parser. It was too slow for long sentences and had been designed for newspaper English, not chat, so I did not include a parser in ChatScript, preferring to build in a POS tagger instead (still a work in progress). One can do some pseudo-parsey things with ChatScript, however.

Suppose you get input like *John is taller than Mary and Mary is taller than Sue. Who is shorter, John or Sue?* Anyone can write rules to handle that specific question. The trick is to write few rules that are general. The following esoteric use of ChatScript illustrates principles and functions for behaving a bit like a parser. The goal is to handle incoming data in one or more sentences, across one or more volleys, and locally handling pronoun references. This includes as input:

Tom is taller than Mary who is taller than Sarah.
Tom is fatter than Joan but she is thinner than Harry.
Tom is taller than Mary but shorter than Sarah.
Tom is taller than Mary. Sarah is shorter than Tom. ...

The code below handles acquiring the facts (not answering the questions) but organizes the data for easy retrieval. It aims for generality. The tricks involve how ChatScript manages pattern matching. Stage one is to preprocess the input to mark in the dictionary where every word and ALL of its concept and dictionary inheritances occur in the sentence. So if word 1 of the sentence is "tiger", then tiger and animal and mammal and ~animals and ~zoo might all get marked as occurring at word 1. So when a pattern tries to match, it can find any of those as being at position 1. But script can also mark word positions, and it can unmark them as well. So the FACTER rule matches a series of items and after creating a fact of them, erases their mark so a rescan of the same rule can try to find a new series of items.

```
concept: ~extensions (and but although yet still)
concept: ~than (then than)

topic: ~compare_test system repeat ()

# set local pronoun
s: ( ~propername * [who she he]) refine()
a: ( _~propername *~2 _who )  $$who = '_0
a: ( _~femalename * _she ) $she = '_0
a: ( _~malename * _he )  $he = '_0

# resolve pronouns
s: ( [who she he]) refine()
a: ( _who $$who) mark(~propername _0)
a: ( _he $he) mark(~propername _0)
a: ( _she $she) mark(~propername _0)

s: ( _~propername * ~propername *~2 _~extensions {be} {less more} ~adjective ~than
~propername)
mark(~propername _1 ) $$and = '_0
```

```
        #! Tom is more tall than Mary
        #! Tom is taller than Mary and Tom is shorter than Joan.
        #! Tom is less tall than Mary
        #! Tom is taller than harry but shorter than Joan.
        s: FACTER ( _~propername {be} {more}  _{less least}  _~adjective ~than
        _~propername )
        $$order = 1
        if (_1) { $$order = $$order * -1 } # flip order
        if ($adj)
        {
           if ($adj != _2 ) # they differ
         {
         if (query(direct_svo _2 opposite $adj ) ) # its the opposite
         {
         $$order = $$order * -1 # flip order
         }
         else {$adj = null} # accept new adjective
         }
        }

# adjust pronouns
if (_0 == who) { _0 = $$who}
else if (_0 == he) { _0 = $he}
else if (_0 == she) { _0 = $she}
else if (_0 ? ~extensions) { _0 = $$and}
if (_2 == who) { _2 = $$who}
else if (_2 == he) { _2 = $he}
else if (_2 == she) { _2 = $she}

if (!$adj)
{
  $adj = _2
 if ($$order == 1) { ^createfact(_0 $adj _3)}
 else {^createfact(_3 $adj _0)}
}
else # already have an adjective to run with
{
 if ($$order == 1)
 {
 ^createfact(_0 _2 _3)
 }
 else
 {
 ^createfact(_3 $adj _0)
 }
}
```

```
unmark( ~propername _0)
unmark(~adjective _2 )
unmark(~propername _3 )
$$who = _3
retry()
```

A responder for handling questions given 2 people is:

```
#! who is taller, Tom or Harry?
#! Of Tom and harry, who is taller?
#! who is less tall, Tom or Harry?
#! who is the taller of Mary and Harry
#! Of Tom and harry, who is least tall?
?: COMPARE ([which who what] be {the} {more most} _{less least} _~adjective < *
_~propername {and or } _~propername)
# 0=less 1=adj 2 = person1 3 = person2
$$order = 1
if (_0) { $$order = $$order * -1 } # flip order
if ($adj)
{
 if ($adj != _1 and query(direct_svo _1 opposite $adj ) ) # they differ
  {
 $$order = $$order * -1
 }
}

# find if _3 is more than _2
nofail(RULE eval(query(direct_vo ? $adj _2))) # who is taller than _2
_7 = @0subject
loop()
{
 if (_7 == _3) {FAIL(rule)} # now matched
 query(direct_vo ? $adj _7) # who is taller than this
 _7 = @0subject
}

if ( $$order == 1) # normal order
{
  if (@0subject) {_3 is.}
  else {_2 is.}
}
else # inverse order
{
 if (@0subject) {_2 is.}
 else {_3 is.}
}
```

The system builds facts in a specific order, like (X taller Y) and if a shorter comes first it flips the order and rewrites using common adjective notation.

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 substitutes.txt file consists of 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 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.

The allownonwords.txt file is a list of non-words that the script compiler should allow in a pattern or output w/o warning you they may be misspellings.

Server/LINUX

While the system comes prepped to run under Windows, it can compile and run under LINUX. The file LinuxChatScript is a file you can set to executable and run. To build it from scratch, follow the steps in LinuxCompile.bat . Server data is logged into serverLog.txt, under control of *:serverlog*

Running the Server

When you run the LINUX program, it defaults to server mode, port 1024. The Windows version defaults to standalone mode.
Either LINUX or windows versions accept the following command line args:

```
ChatScript port=xxxx                # be a server on this port
ChatScript local                    # run locally, not as a server
ChatScript client=ip:port           # be a client to test a remote server
ChatScript client=localhost:port    # be a client to test a local server
```

There are batch files server.bat (be a server on the default port), localClient.bat (be a local client). Remember that to be a remote server you need to make your port available for inbound TCP if you have a firewall- if you don't have a firewall you must be insane.).

If the engine is not allowed to alter the server machine, you can start it with the parameter "sandbox" (see sandboxserver.bat) which disables Export and System calls.

Communicating with the Server

The client webpage/program connects on a socket (or http webpage) to the ip and port of the server. The message it sends is a concatenation of three null-terminated text strings.

The first string is the user login name. The second is the id of the chatbot to talk to. If this is a null string, the system will connect to the default bot. The third string is the message. If the message is null, this is a start of new conversation. This MUST be the first thing you do with a new user. Ideally you do it whenever a new conversation is starting with that user. The message sent to the server during a conversation should never be null (since that looks like a conversation start). Either always prepend a blank on every line from the user, or add a blank if the user presses ENTER without anything else.

The server shuts down the connection after each volley, so you have to reinitiate a connection for each volley. Normally you send the "start of new conversation" only on the first of many volleys. As long as the user is connected to the webpage, for example, you wouldn't send it again.

The chatbot can wait forever for each input (the connection is terminated for each volley) and the only way to know that the human "left" is when the human "comes back" with a start of a new conversation.

If you run the Windows ChatScript engine with client=ip:port as an argument, it will act as a client to talk to and test the server. The ip address must be numeric and the :port is optional. The client will start a conversation and then loop with you conversing to it.

Testing the Server

You can test everything on your own machine in Windows. Launch the server by double clicking on server.bat and then launch the client by double clicking on localclient.bat. Since this is on your own machine, firewall opening the port is unnecessary.

For LINUX, just perform the equivalent commands of the batch files (except that since you can't readily run multiple apps, you'll have to background the server presumably by doing a nohup command on it.

Preparing for compiling on the Server

I develop the source on a Windows machine and transfer it to a LINUX box. To insure the source does not have carriage returns, I use *:clean* to read and write all src directory files without carriage returns.

Unique names using a server

When you have a server, the system assumes the user names will be unique (comes from a private system). If your server is public, people's names can collide and the system will think both people are the same. If you send the server the user's name with a period in front of it, the server will automatically prepend the name with the user's IP address, keeping them unique. Of course, if the user eventually comes back some other day on a different IP address, they lose their history. Can't be helped. An alternative is to ignore their login name and log them in as "guest", which will append their ip address to that.

Testing for server presence

If you send the message: null 1 null (that's the null string user id, the string of the character "1" as bot id, and the null string message, the server will send back the string of the character "1", with no logging done and minimal load on the server. This constitutes an echo-test to prove the server is running.

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.

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.

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 noerase 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?

It 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 ^noerase() and you can make the topic not allow responders to erase with noerase 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.

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)
```

AutoGeneration of Responders

Stylistically, I write a lot of topics which more or less alternate between asking a question and providing the corresponding answer for the chatbot. E.g.

```
topic: ~food […]
t: What do you typically have for lunch?
t: I usually have a sandwich for lunch.
t: Where do you usually have lunch?
t: I eat lunch at college.
```

In such cases I usually want the chatbot to be able answer the question directly if asked of it and, of course, then not volunteer the answer again. To do this I have to write responders that match the question and then ^reuse the answering gambit (after first labeling it for reuse). This back and forth is tedious, as is writing the responders. So I've semi-automated the process.

You can attach special comments (#ai) to automatically generate reasonable responders. You can use this to reuse a gambit or to generate a completely new responder. There are a few ground rules.

First, each rule must start on a new line, though it can be indented in that line. This is good practice anyway.

Second, the data must be in a file labeled *.ai. If you give the :generate command and name filename (with or without the .ai) the system will generate a corresponding file named *.top, comprising both your ordinary script and newly generated script.

Third, the special rule begins with #ai and fits on a single line. If there is an = and a sentence BEFORE the =, that is the question to respond to. If there is nothing before the =, that means use the input from the gambit before the current one. If there are words AFTER the =, that is the answer to give in response to the before. If there are no words AFTER the =, that means reuse the label of the most recent gambit. If you have nothing before or after the =, you may omit it.

```
t: What do you typically have for lunch?
t: WHATLUNCH() I usually have a sandwich for lunch.
#ai
#ai =
#ai What is your usual lunch? =
#ai What do you eat for lunch? = I eat sandwiches.
```

The above illustrates the possibilities. The first two #ai's say to use the prior gambits output (*What do you typically have for lunch?*) and reuse the answer from the current labeled gambit (*WHATLUNCH).* The second #ai says the question to respond to is *what* *is your usual lunch?*, again reusing WHATLUNCH. The third #ai creates a new question *What do you eat for lunch?* And a new answer *I eat sandwiches.*

The system automatically builds a semi-generalized pattern where the words of the question are encased in << >> so they can occur in any order. And, the system ignores what it considers to be useless words. It also automatically flips any word with a canonical form listed in LIVEDATA/canonical.txt and uses the canonical form (so pronoun your becomes you).
And it takes LIVEDATA/generalizer.txt and uses the pairs there to switch. So if your question was *I like wine*, the file might generalize *like* to ~like which are a collection of synonyms for like. If you generalize to null, the system ignores the word. I do this for "a", which means all the determines *a,an,the* disappear.

Esoterica and Fine Detail

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.