ChatScript Basic User's Manual

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

Revision 5/3/2015 cs5.23

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.

In most common usage, what you build is a “chatbot”, a program that takes input from a human and outputs a response. It converses with you. But I have also built a chatbot that is dedicated to reading a document. The input, in that case, is the entire document. What, if any, output it generates is up to its program. My program was designed to find the table of contents, each chapter boundary, and where the index at the back was.

So, more broadly, ChatScript is a system for manipulating natural language, not just for building a chatbot.

This document is about how to write script. There is a more general discussion about how to think about authoring a bot in the paper *Paper- Writing a Chatbot* in the documentation. It also has a quick 2 page guide to blindly creating a simple chatbot without reading much of this document.

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 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 a topic. It calls engine functions and conditionally executes other topics.

Whitespace: ChatScript generally ignores excessive white space. 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 words in patterns should be lower case unless they are proper names or the word "I". Don't make a lowercase word be upper case in a pattern merely because you think of it as starting a sentence.

Comments: the comment character is the hash mark. It extends to the end of the line. Ordinary comments must have at least one space after them, to allow the system to distinguish comments from numeric constants like : #NOUN

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

There are special documentation comments that start with #! described elsewhere.

Legal declaration names: A topic or function or variable name must contain only alpha-numeric characters, underscores, hyphens, and periods. They must their special character for their type and then continue with a starting alphabetic character.

DEMO

Hello Word

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

WINDOWS: Download and extract on a windows system into a directory (mine is called ChatScript), keeping files in their respective folders. Double click on the chatscript.exe file.

LINUX: To run on a Linux system is not really much different. Download and extract into a directory in Linux, keeping the files in their folders. The executable that ships with the product is LinuxChatScript32. You need to alter permissions to make it executable. And it's a 32-bit version, so if you are running on a 64-bit system you need to insure you have the 32-bit libraries installed via:

sudo apt-get install ia32-libs

Also, the Linux version defaults to server mode, so you should run it as:

./LinuxChatScript32 local

1. The system prints out a bunch of statistics as it loads data. After which is says: “Enter user name”.
2. 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.
3. 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.* Actually, you can say or ask anything and get an almost reasonable response because Harry has *quibbles.* If he knows nothing specific on your topic (which is almost universally true), he can stall and quibble instead. So you can have a conversation about anything.

If you try to run the 32-bit version on a 64-bit machine, you may run into issues if your machine distribution has deprecated ia32-libs. To fix this, you need to install them as follows:

```
sudo dpkg—add-architecture i386
sudo apt-get update
sudo apt-get install libc6:i386
sudo apt-get install lib32stdc++6
```

OK. Now you’ve run what comes pre-installed.

If you want to learn about the simple Harry bot and try changing it, read the BOT-HARRY document. Same if you want to build your own bot (starting by cloning Harry).

Let’s quickly survey what comes built in.

Note on file types

ChatScript reads ordinary ascii text files and utf-8 files. It does not read utf-16 files correctly.

Fast Overview of .top files

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 rules. 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 gambit 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, rules come in any order. The *gambits*, t: lines, 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".

```
        Topic: school [school university learn]
        t: Where do you go to school?
                a: (Rochester)That's a great school.
        t: I go to Harvard.
        t: What is your major?
        t: I am studying finance.
```

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 one example of matching input, not all possible inputs that can match. It helps you understand what a responder or rejoinder is supposed to react to. It has no impact whatsoever on a user in chat.

What Files are Where

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 (CS does not use this file, it merely records it), 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 also stored in the topic file.

Dictionary Files: The DICT folder is the underlying dictionary of the system. You probably won't modify it. It has one or more subfolders corresponding to languages or subset dictionaries supported. The default language is ENGLISH.

Dictionary Extension Files: The LIVEDATA folder contains extensions to the dictionary and run-time 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 (#! sample inputs), which allows the system to prove your patterns actually do what you intend, that data is stored in the VERIFY directory after compilation. There are folders for HARRY, ONTOLOGY, WORLDDATA, STOCKPILE, NLTK, POSTGRES and QUIBBLES. Normally when you define a chatbot, you add a folder with the name of your chatbot. That way you can just swallow updates to the main system whenever a new release is created.

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.

LOGS: If ChatScript detects bugs during execution, it stores them in bugs.txt in this folder. A server will also store its log here.

Compilation Files: The folders LINUX, MAC, and VS2010 are for rebuilding the executable engine. LOEBNERVS2010 builds a Loebner contest 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 & filesHarry.txt – what files to compile with :build 0 and :build Harry
filesNLTK – builds a natural language toolkit bot called NLTK
filesPostgres- builds a simple postgres bot- you need postgresql installed

filesStockpile – builds a planner bot called Stockpile
LinuxChatScript32 – Linux 32-bit executable engine (defaults to server mode)
LinuxChatScript64 – Linux 64-bit executable engine (defaults to server mode)
loebner.exe- engine compatible with Loebner competition protocol
changes.txt – list of changes between releases
version.txt – the current version
*.dll - windows dll for PostgreSQL use
talk.vbs – script to enable voice output in windows

lib: server library files for LINUX are kept here.

REGRESS: Files that can do regress tests of various kinds are here.

Server Batch Files: files for windows that can run the engine as a local server or a local client.

TMP: transient files used to support debugging are kept here.

SIMPLE TOPICS

The system does not execute all rules. Nor do you directly choose a rule to execute. Instead, you (the author) organize collections of rules into topics and the system 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:, tablemacro:*, *outputmacro:*, *patternmacro:, dualmacro:, bot:, data:* and *replace:*). A topic name must start with a ~, an alphabetic character, and then be a standard legal name (contain only alpha-numeric characters, underscores, hyphens, and periods).

Keywords

Keywords allow the system to consider this topic based on matches with the user input. If the user input has no keywords in common with the topic, then normally the topic and all its rules will not be considered. 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.

If you are in the middle of some topic and the user says something unrelated, then the system will hunt for a topic that matches keywords of the input and see if some matching topic has a responder to react to the input. If so, the system will switch to that topic. The new topic becomes the "active" topic and the old topic becomes "pending", put on a list of topics to return to when the new topic becomes exhausted. This behavior is the default behavior and you can script other behavior by creating a control script to do so (advanced).

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. But if a rule in that topic is going to be considered, either the topic has keywords that can be found in the pattern of that rule, OR, the topic is already the current topic (some other rule had keywords in its pattern that also were topic keywords and got you entry into the topic) OR, the topic was directly called by the control script or some other topic.

Gambit Rules

In addition to responders for user input (*s: ?: u:*) a topic can have gambits to 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 reuse a gambit it was going to volunteer anyway. It is entirely up to you the order of responders and gambits. You can segregate *s:* from *?:* from *u:* or co-mingle them. You can have responders intermixed with gambits.

What does it mean for the chatbot to be "in control" of the conversation? When the user types in a statement or question, the system will see if it has a rejoinder or a responder that directly matches the input. If it does, it will respond with that. That forced response means the human was in control of the conversation. If, however, the chatbot cannot find a response for the input, then the chatbot is free to say whatever it wants. It is in control. Maybe it will issue a gambit from the topic it was last in. Maybe it will make a gambit from a topic closely related to the user's input. Maybe it will try to steer the conversation to a specific new topic. All that is controlled by the logic of the control script (of which there is a default one supplied with ChatScript but which you can author yourself).

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 a theme. 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. Similarly, a responder that reacts to an input will give its message and then erase itself. If the user repeats his input, that rule cannot respond again, and some other rule will have to answer.

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 filesHarry.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 rejoinders 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.

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. Pattern matching a sequence is limited to 5 words in row and will do both original and canonical forms.

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.
```

Since the words can be matched in any order, this resets the scanning mechanism back the original starting condition, which is always < * meaning you can match the next the next item anywhere in the sentence.

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 consisting of only alpha-numeric characters and underscores. A concept has a list of words it defines. 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 1600 of them), the system automatically defines a bunch of dictionary-based sets. These include:

parts-of-speech like *~noun* (see POS-Tagging manual)
~number and *~placenumber* (like 1st, second)
~money (like $24) and *~timenumber* (like 4:32)
~propername, ~malename, ~femalename, ~humanname, ~firstname
~childword, ~adultword, ~unknownword
~timeword (refers to time), *~spacename* (refers to a place), *~url*
~dateinfo (combinations of month with day and/or year)
~qwords (question starting words like who what where when why how)
~topic (names of topics with or without ~ in front)

Capitalization

However you type your input, with or without capital letters, the system tries to figure out the word you actually mean. If you write “Can I go back”, you don't mean that “Can” is a proper noun. You mean “can I go back”, and so that is what the system sees. This means when you write patterns, you do not write capital letters on first words of sentences. You only write them for proper nouns. So this pattern:

u: (Can I go back)

is wrong and will not match anything and should be

u: (can I go back)

Interjections, “discourse acts”, and concept sets

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

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

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

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

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

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

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

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

"~repeatme","~repeatinput1","~repeatinput2","~repeatinput3","~repeatinput4","~repeatinput5","~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.

Floating point numbers migrate to integers if they match value exactly, while currency values become floating point.

Personal pronouns like *me*, *my*, *myself*, *mine* move to the subject form *I*, while *whom, whomever, whoever, whose* shift to *who* and *anyone somebody anybody* become *someone* and *whatever* becomes what*, whenever* becomes *when*, *whichever* becomes which.

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

The same is not true for concepts:
u: ('~extent_adverbs)
This is not legal, because there is no way for the system to determine if the set was matched on the original or canonical form of the sentence.

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_.
If you want to test for a sequence of optional words, you can do two things:
u: ({are} {you}{going} home)   or   u: ({“be you go”} home)
In a quoted phrase, you can use either all canonical or all exact forms. And you can do a quoted phrase of 4 words long (but not longer).

Commands

You can issue commands to the system (prefix is colon) to inquire about things, control things, debug things, etc. In this simple section, we look at commands to examine 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. A list of all commands can be gotten by typing **:commands**. Documention on most of them is in the debugging manual.

: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 :word 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. **:down entity 1** or **:down ~animals 2**

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?

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. *I like you?* and *I like   you   ?* print the same on output

If you actually need to control spacing, consult "formatted double quotes" in the advanced manual.

Literal Output \

To output characters that have reserved meaning to the engine, like [and], you need to put a backslash immediately 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?*

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. The ultimate variable is the fact, but that has its own manual.

_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. Whenever you start a volley, you should presume match variables all hold unknown junk.

You are allowed _0 through _19. I often use the _10 to _19 range as “safe” variables for the duration of a volley, because I will never match that many variables in a single sentence. I am unlikely to even match more than 5. So I can dedicate them any way I want to.

When the system memorizes your underscore match, it stores both the original word, its canonical form, and the position of the text. 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 I 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.*

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. A null variable prints nothing on output.

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

```
    s: (  _~fruit [_~animal _bear] _~like)
```

In the above, _0 is a fruit and _1 is a like, and the _~animal is discarded when the [] completes. If you wanted the value kept, you would have to put it outside the [] as follows: s: (_~fruit _[~animal bear] _~like)

$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 an alphabetic letter and then the rest must be legal name characters. You initialize it using a C-style assignment in the output. The = assignment operator MUST be separated from the variable and the value by at least one space, otherwise the system has no way to tell you don't want it to simply output some bizarre word.

Unlike match variables, user variables hold a single value only.

```
	s: ( I eat _*1 >) $food =  '_0  I eat oysters.
```

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: ( I eat _*1 >)
		$food =  '_0
		I eat oysters.
```

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 $$ at the start, e.g. *$$food.*

In addition to simple assignment, you can do +=, -=, *=, /=, %=, |=, &=, ^=, and |^=, eg

```
	$$myvar += 100.
```

|^= turns off bits and $x |^= 2 is equivalent to $x &= (-1 ^ 2)

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

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

So be careful with extended arithment. Each operation applies to the result of the last. $myvalue += 2 * 4 means ($myvalue + 2) * 4.

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

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

You cannot use the name of the variable you are assigning to in the right hand side, as arithmetic is performed progressively across the terms, so the variable is overwritten on the first term. Similarly, you may be surprised by something like this:

```
	$myvar -= $articlesize * 10
```

because it first deducts $articlesize from $myvar and then multiplies that result by 10.

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 if *$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. Relational tests use no whitespace, they are all one big token. One such relation is =, also writeable as ==.

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

Here, the rule will only start checking for input matches if *$gender* has the value *male (*case insensitive). If it is not defined or has any other value, this rule fails immediately. A relational test requires the two sides of the relation and the relation symbol all be jammed together with no spaces. So the following rule is tantamount to seeing if *$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.
```

For equality testing you can use the != operator.

```
    ?: ( $var!=5) OK.
```

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

```
    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 with position, the assignment is dual form and passes the position along.

$variables read by system

$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
$botprompt – message for console window to label bot output
$userprompt – message for console window to label user input line
$crashmsg – message to use if a server crash occurs.
$token- bits controlling how the tokenizer works. By default when null, you get all bits assumed on. The possible values are in src/dictionarySystem.h (hunt for $token) and you put a # in front of them to generate that named numeric constant.
$abstract – topic used by :abstract to display facts if you want them displayed
$prepass – topic used between parsing and running user control script. Useful to supplement parsing, setting the question value, and revising input idioms.

Variables set by system
$randindex- the current random seed for this volley

Clearing variables

You can erase a user variable or match variable by setting it to null:

$myvar = null
_3 = null

%System Variables

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

Date & Time & Numbers

%date – one or two digit day of the month
%day - Sunday, etc
%daynumber – 0-6 where 0 = Sunday
%fulltime – seconds representing the current time and date (Unix epoch time)
%hour – 0-23
%leapyear – boolean if current year is a leap year
%daylightsavings – boolean if current within daylight savings
%minute – 0-59
%month - 1-12 (January = 1)
%monthname – January, etc
%second – 0-59
%volleytime – number of seconds of computation since volley input started.
%time – hh:mm in military 24-hour time
%week - 1-5 (week of the month)
%year – e.g., 2011
%rand – get a random number from 1 to 100 inclusive

User Input

%bot – current bot responding
%revisedinput – Boolean is current input from ^input not direct from user
%command – Boolean was the user input a command
%foreign – Boolean is bulk of the sentence composed of foreign words
%impliedyou *– Boolean was the user input having you as implied subject*
%input – the count of the number of volleys this user has made ever
%ip – ip address supplied
%length – the length in tokens of the current sentence
%more – Boolean is there another sentence after this
%morequestion – Boolean is there a ? or question word in the pending sentences
%parsed – Boolean was current input parsed successfully
%question – Boolean was the user input a question – same as ? in a pattern

%quotation – Boolean is current input a quotation
%sentence – Boolean does it seem like a sentence (subject/verb or command)
%tense – past , present, or future simple tense (present perfect is a past tense)
%user – user login name supplied
%userfirstline – value of %input that is at the start of this conversation start
%userinput – Boolean is the current input from the user (vs the chatbot)
%voice – active or passive on current input

Chatbot Output

%inputrejoinder- rule tag of any pending rejoinder for input or 0 if none
%lastoutput – the text of the last generated response *for the current volley*
%lastquestion – Boolean did last output end in a ?
%outputrejoinder- rule tag if system set a rejoinder for its current output or 0
%response – number of responses that have been generated for this sentence

System variables

%all – Boolean is the :all flag on? (:all to set)
%document – Boolean is :document running
%fact – Numeric value most recent fact id
%freetext – kb of available text space
%freedict – number of unused dictionary words
%freefact - number of unused facts
%regression – Boolean is the regression flag on
%server – Boolean is the system running in server mode
%rule – get a tag to the current executing rule. Can be used in place of a label.
%topic – name of the current "real" topic . if control is currently in a topic or called from a topic which is not *system* or *nostay*, then that is the topic. Otherwise the most recent pending topic is found.
%actualtopic – literally the current topic being processed (system or not).
%trace – Numeric value of the trace flag (:trace to set)

Build data:

%dict -   date/time the dictionary was built
%engine – date/time the engine was compiled
%os – os involed (linux windows mac ios)
%script – date/time build1 was compiled
%version – engine version number

You actually can assign to any of them. This will override them and make them return what you tell them to and is a particularly BAD thing to do if this is running on a server since it affects all users. Typically one does this as a temporary assignment in a #! comment line to set up conditions for testing using *:verify.* Making them return a new value is NOT the same thing as making the engine have a different value. Unless the variable is marked as settable, setting a value affects only the value returned by a future call to the system variable. It does not change engine values the variable is meant to reflect.

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 and read it in under a :build command. Go read the facts manual for more about facts.

Summary

Not as simple as you might have wanted. Doubtless I told you much more than you really wanted to know at this stage, but it sets the scene so you know there is a lot of capability there (though truly we have barely scratched the surface here). Just remember, to start, all you need is to write a topic, with keywords, trivial gambits, responders and rejoinders with simple patterns, and output that is simply exactly what you want the bot to say.