ChatScript Basic User Manual

Copyright Bruce Wilcox, mailto:gowilcox@gmail.com www.brilligunderstanding.com
Revision 11/10/2018 cs9.8

- Overview
- Simple Topics
- Simple Patterns
- Simple Output
- Variables
- Summary

OVERVIEW

ChatScript (CS) 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 Writing a Chatbot in the documentation.

Input and output sentences

ChatScript (CS) takes one or more *input sentences* from the user. A *sentence* in CS nominally means the sequence of words up until some sentence *terminal token* chars. By default these characters end sentences:

dot	questionmark	exclamationpoint	semicolon	colon	hyphen
.	?	!	;	:	-

E.g. the following user phrase is split in three sentences:

```
What's your name? My is Alfred; I'm from London.
```

One can also disable sentence ending.

ChatScript outputs one or more sentences back to user.

```
My name is Harry!
I'm a chatbot demo, from California (USA).
```

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 chatbot 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.

Code Script Syntax

CharSet Encoding

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

Whitespace

ChatScript generally ignores excessive white space. You can have a plethora of tabs, spaces, and newlines.

E.g. the rules:

```
?: (When will you die) I don't know.
```

and:

```
    ?:  ( When   will you    die )
        I don't know  .
```

are interpreted in the same way.

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".

NOTE: 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`

```
# this line is a comment
s: (I hate meat) So do I. # a comment at the end of an output statement
```

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

```
#! I hated meats
s: (I hate meat) really?
```

Legal declaration names

A *topic* or *function* (defined later) must contain only alpha-numeric characters, underscores (`_`), hyphens (`-`), and periods (`.`). They must begin with tilde (~‘) and then continue with a starting alphabetic character.

Variables must start with `$` or `$$` or `$_`, then continue with a starting alphabetic character and then continue with alpha-numerics, underscores, or hyphens.

Hello Word Demo

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 in BINARIES folder.

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 `LinuxChatScript64` (64-bit version) in BINARIES folder. You need to alter permissions to make it executable. Also, the Linux version defaults to server mode, so you should run it as:

```
BINARIES/LinuxChatScript64 local
```

This will NOT work if you have a 32-bit Linux and you will need to recompile on your machine. See `Installing and updating ChatScript` manual.

MacOs With MacOS the executable file changes:

```
BINARIES/MacChatScript local
```

The system prints out a bunch of statistics as it loads data. After which is says:

```
Enter user name:
```

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 the message:

```
Welcome to ChatScript
```

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.

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 BOTHARRY document. Same if you want to build your own bot (starting by cloning Harry). Let's quickly survey what comes built in.

Fast Overview of topic files (sample: `RAWDATA/HARRY/childhood.top`)

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:

rule type	syntax	description
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.

rule type	syntax	description
rejoinders	`a:b:c:d:e:`	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	`r:t:`	are the story the chatbot wants to tell on a subject or the conversation the chatbot is trying to steer the user into. `r:` are random gambits (explained later).

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`. 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 for 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 a file a `#` issued only once on a line 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

Directory	files
BINARIES	Executable Files. All executables are in the BINARIES folder. The system assumes they are launched from there (is the current working directory) and changes directory up a level to access all the other folders.
USERS	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.
DICT	Dictionary Files. The DICT folder is the underlying dictionary of the system. You probably won't modify it. It has a subfolder ENGLISH which is a full dictionary it uses normally. If you are trying to run on a mobile device, you probably want to copy over the contents of BASIC into ENGLISH, to use a smaller dictionary
LIVEDATA	Dictionary Extension Files. The LIVEDATA folder contains extensions to the dictionary and run-time system that you might change as an advanced author.
RAWDATA	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). 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.
TOPIC	That data is run through the script "compiler" and the output is stored in the TOPIC directory, which holds your compiled script data.
VERIFY	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.

Directory	files
SRC	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.
WIKI	Documentation Files. This document and others can be found in WIKI.
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 `changes.txt` - list of changes between releases `version.txt` - the current version `talk.vbs` - script to enable voice output in windows
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.
Third Party Tools	reference to some ChatScript third party tools.
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:`, `canon:`, `query:`, `plan:`, `describe:`, and `replace:`).

A topic name must start with a `~`, an alphabetic character, and then be a standard

legal name (contains 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.

If you created .top files for various topics with rules and dropped your file into the HARRY folder
and rebuilt the bot (:rebuild Harry), 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 want to date Olive Oyl?

        b: ( no ) Me neither.
        b: ( yes ) She was tall for a woman.

    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.

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 and you want to add a label, 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.
```

Note, ChatScript is a token-oriented language (tokens generally being a collection of characters without spaces). You need to put one or more spaces after the `:` and before the label. `?:EYECOLOR` is a single token and not a declaration of a rule and a label.

Also, while you can use any case you want in the rule label, internally ChatScript will convert it to full uppercase. Routines that deal with rule labels know this so you don't have to care. All uppercase rule labels stand out more easily in traces and in script source and are considered good form.

The `childhood.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.

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 `*n`

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, will there be anyone at home?* nor will it accept *when you went home* which hasn't room for the first `*1`.

Range-restricted Wildcards `*~n`

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.

Position is freely reset to the start following the `<< >>` sequence so if you had the pattern:

```
u: ( I * like << really >> photos)
```

and input *photos I really like* then it would match because it found `I * like` then found anywhere *really* and then reset the position freely back to start and found *photos* somewhere 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"]
```

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 represent 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` as well as general open concepts like `~number`.

For a full list, see the ChatScript System Variables and Engine-defined Concepts manual.

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 )
```

Contractions

ChatScript automatically expands contractions. DO NOT put contractions in your patterns. You will break the expansion of that contraction.

Proper names

You should always put multiple-word proper names in double quotes, particularly ones with embedded punctuation. You want CS to know that the entire phrase is considered a single entity. So

```
u: ( "Dr. Watson" )
u: ( "The Beatles" )
```

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 `interjections.txt` file (for English language, see interjections.txt). Also there are some in the `texting.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 either come through as the user input instead of what they wrote or will

merely be marked as interjections, leaving the words unchanged (see interjections in `Practicum - Spelling and Interjections`).

For example *yes* and *sure* and *of course* are all treated as meaning the discourse act of agreement in the interjections file. If you don't suppress interjection word remapping, 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 listed in the ChatScript System Variables and Engine-defined Concepts manual.

If interjections at the start of a sentence are broken off into their own sentence (as they used to be by default but are no longer), 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.

But you can turn off this behavior and then the above pattern works perfectly well.

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 true for concepts:

```
u: ( '~extent_adverbs )
```

Not ! And NotNot !!

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.
```

While `!` Checks for the entire rest of the sentence, `!!` checks just the next word from where you are. So

```
u: ( test !!this)  I win
```

Will match input *test that this works* and not match *test this works*.

Optional Words { }

Sometimes you can expect a word might or might not be supplied. Your pattern can reflect this and swallow 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 not 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 ChatScript 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*,

```
>>:up temple

For temple:
 Set hierarchy:
    ~religious_buildings
       ~building
```

```
         ~tool
         ~locatedentity
            ~there
         ~immovable
            ~functions
         ~enterable
         ~artifacts
            ~objects
               ~nounlist
            ~human_data
Wordnet hierarchy:
 temple~1:N    the place of worship for a Jewish congregation
   is house_of_worship~1  any building where congregations gather for prayer
      is building~3 a structure that has a roof and walls and stands more or less permanent
         is structure~1 a thing constructed
            is artifact~1 a man-made object taken as a whole
               is whole~1 an assemblage of parts that is regarded as a single entity
               ~entirety_words
               ~goodness
                  is object~1 a tangible and visible entity
                  ~verbal_disagree
                     is physical_entity~1 an entity that has physical existence
                        is entity~1 that which is perceived or known or inferred to have it
                        ~nounroot
 temple~2:N    place of worship consisting of an edifice for the worship of a deity
   is house_of_worship~1  any building where congregations gather for prayer
 temple~3:N    an edifice devoted to special or exalted purposes
   is building~3  a structure that has a roof and walls and stands more or less permanently
 temple~4:N    the flat area on either side of the forehead
   is feature~2  the characteristic parts of a person's face: eyes and nose and mouth and c
   ~focus
      is body_part~1 any part of an organism such as an organ or extremity
         is piece~11 a portion of a natural object
         ~unitmeasure
            is thing~1 a separate and self-contained entity
               is physical_entity~1 an entity that has physical existence
```

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 ChatScript Advanced Output 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` and tab with `\t`.

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 ChatScript Fact Manual 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 `_30`.

I often use the above `_10` 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, they are allocated in the order of the pattern. E.g.,

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

In the above, `_0` is a fruit and `_2` is a like, and `_1` is `_~animal` or `_bear`.

If you had NOT put _in front of bear, you are at risk that the `~like` match may be `_1` or `_2`, depending on what happened inside `[]`. That's your headache if you use nested memorization.

Be aware that this expression:

```
u: (_1)
```

does not memorize the input word "1". It instead names match variable 1 and asks if it has a value. If you were desperate to memorize the word 1 (how weird), you'd have to do this:

```
u: ( _~number _0=1 )
```

$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 or two and then an alphabetic letter and then the rest must be alpha, digit, underscore, or hyphen. 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.
```

You can even assign sets of facts in various ways (see ChatScript Fact Manual for understanding facts) like:

```
@2 = @3 # put all facts in 3 into 2
@2 += @3 # augment 2 with facts of 3 (allows duplicates)
@2 -= @3 # remove facts of 3 from 2
@2 = $$factid # take a fact id and put it as the only content of 2
@2 += $$factid # add fact id to 2
@2 -= $$factid # remove factid from 2
```

WARNING:
Be careful with extended arithmetic. Each operation applies to the result of the last.

```
$myvalue += 2 * 4 means ($myvalue + 2) * 4.
$tmp = %hour + $tmp means (2 * %hour)
```

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

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

You should not 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 ) That's less controversial these days.
```

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.

Clearing variables

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

```
$myvar = null
_3 = null
```

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 ChatScript Fact Manual for more about facts.

% System Variables

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

See ChatScript System Variables and Engine-defined Concepts.

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.

[Wiki home] - [Advanced User Manual]