ChatScript Advanced User's Manual

Revision 8/18/2019 cs9.62

This manual is a grab bag of various capabilities. There are separate other Advanced manuals like Advanced Topic Manual, Advanced Pattern Manual, Advanced Concept manual, and Advanced Variable Manual for in-depth extras on main CS ideas.

Review: Overview of how CS works

CS is a scripting language for interactivity. Each time CS communicates with the user, this is called a *volley*.

Volleys are always asynchronous. In CS, each volley actually consists of accepting an incoming input from an arbitrary user, loading data about the user and their state, computing a response, writing out a new state, and sending a response to the user.

Topics and Rules

The fundamental code mechanism of ChatScript is the topic, which is a collection of rules.

Rules have pattern and code components.

Within a topic each rule is considered in turn by matching its pattern component. Patterns can access global data and the user's input, can perform comparisons, and can memorize sections of input data.

If the pattern fails, the next rule in the topic is considered. If a pattern succeeds, the rule's code section is then executed to completion (barring error conditions).

A rule's code can be a mixture of CS script to execute and words to say to the user.

Code can invoke other topics or directly request execution of a specific rule. When the rule code completes, if user output has been generated, then by default no more rules are initiated anywhere in the system. Rules currently in progress complete their code. If no output was generated, the topic continues on to the next rule, trying to match its pattern. When a topic completes without generating output, it merely returns to its caller code, which continues executing normally.

Rejoinders

So how is it that CS handles returning input from the user? A rule that generates user output may have rules called rejoinders that immediately follow the rule.

Rejoinders are intended to analyze the specific next input from the user to see if certain expectations are met and decide what to do. If, for example, we output a yes or no question, one rejoinder rule might look for a yes answer, while another rejoinder hunts for a no answer.

When CS outputs text to the user, if the rule has rejoinders, CS notes the rule. When new user input arrives, CS will try executing the rejoinder rules immediately, to see if they match the user's input. All previous stack-based functions are gone, all previous stack-based calls from other topics are gone.

CS is just in the here and now of this topic and the rejoinders of that rule. If CS finds a matching rejoinder rule, it continues in this topic. If it doesn't, CS reverts to globally using whatever the control script dictates it try for any user input.

User variables

In addition to script code, ChatScript has data. It supports global user variables whose names always start with `$`, e.g., `$tmp`. Global means they are visible everywhere. You don't have to pre-declare them. You can directly use one and you can just summon one into existence by assigning into it:

```
$myvariable = 1 + $yourvariable
```

`$myvariable` is created if it doesn't already exist. And if `$yourvariable` hasn't been created, it will be interpreted as 0 or `null` depending on context (here it is 0).

User variables always hold text strings as values.

Numbers are represented as digit text strings, which are converted into binary formats internally as needed.

Text comes in three flavors.

First are simple words (arbitrary contiguous characters with no spaces).

Second are passive strings like *meat-loving plants*.

Third are active strings (which you haven't read about yet) like:

```
^"I like $value"
```

Active strings involve references to functions or data inside them and execute when used to convert their results into a passive string with appropriate value substitutions.

Other languages would name a CS active string a format string, and have to pass it to a function like sprintf along with the arguments to embed into the format. CS just directly embeds the arguments in the string and any attempt to use the active string implicitly invokes the equivalent of sprintf.

User variables also come in permanent and transient forms.

variable scope	syntaxexample	description
permanent	`$permvar`	start with a single `$` and are preserved across user interactions (are saved and restored from disk). You can see and alter their value from anywhere.
transient	`$$transientvar`	start with `$$` and completely disappear when a user interaction happens (are not saved to disk). You can see and alter their value from anywhere.
local	`$_localvar`	(described later) start with `$_` and completely disappear when a user interaction happens (are not saved to disk). You can see and alter their value only within the topic or outputmacro they are used.

System variables

System variables begin with %. Normally these are simply read-only data, but it is legal to assign to them as well, with certain consequences.

```
%response = 5
```

The first consequence is that the change is global, across all bots and users, whether the system is stand-alone or a server.

The other consequence is that usually the change is locked in permanently until you tell the system to release it by assigning a dot to it.

```
%response = .  # release current override and use the normal value again
```

Some assignments are not locking. %input is one of those.

In addition to overriding system variables, if "regression" via

```
%regression = 1
```

is turned on, some variables return fixed values. Things like date and time have a constant value so as not to interfere with regression testing.

Facts

ChatScript supports structured triples of data called facts, which can be found by querying for them. The 3 fields of a fact are either text strings or fact references to other facts. So you might have a fact like

```
(I eat "meat-loving plants")
```

and you could query CS to find what eats meat-loving plants or what do I eat. Or even more generally what do I ingest (using relationship properties of words).

JSON data returned from website calls are all represented using facts so you can query them to find the bits of data you seek.

Like user variables, facts can be created as *transient* or *permanent*.

Permanent facts are saved across user interactions, transient ones disappear automatically. When you want to point a user variable at a fact, the index of the fact is stored as a text number on the variable.

Output

Some of the text in rule output code is intended for the user. There is pending output and committed output.

Pending output consists of whatever isolated words that are not part of executing code exist in the code. They accumulate in a pending output stream, and when the rule finishes successfully, the output is committed. If the rule fails, the pending output is canceled.

You can also make function calls that directly commit output regardless of whether the rule subsequently fails.

Marking

When CS receives user input, it tokenizes it into sentences and analyzes each sentence in turn. It "marks" each word of the sentence with what concepts it belongs to.

Concepts always begin with ~.

Usually concepts are explicit enumerations of words, like `~animals` is a list of all known animals or `~ingest` is a list of all verbs that imply ingestion.

Sometimes concepts are implicit collections handled directly by the engine, like `~number` is the implied set of all numbers (we wouldn't want to actually enumerate them all) or `~noun` is the set of all nouns or `~mainsubject` is the current subject of the sentence.

After this marking analysis, patterns can efficiently find whether or not some particular concept is matched at a particular position in the sentence.

CS actually analyzes two streams of input, the *original* input of the user and a *canonical* form of it. So the system marks an input sentence of *my cat eats mice* and also marks the parallel sentence *I cat eat mouse*, so patterns can be written to catch general meanings of words as well as specific ones.

Memorizing

Rule patterns can dictate memorizing part of the input that matches a pattern element. The memorized data goes onto "match variables", which are numbered `_0`, `_1`, ... in the order in which the data is captured.

CS memorizes both the original input and the canonical form of it. The pattern can use match variables in comparisons and the output can also access the data captured from the input.

Control flow & errors

CS scripts execute everything as a call and return (no GOTO).

The return values are the current pending output stream and a code that indicates a control result. That result in part affects how additional rules in the calling topics or functions execute, in that you can make a rule return a failure or success code that propagates and affects the current function, or rule, or topic, or sentence, or input.

So a failure or success down deep can, if desired, end all further script execution by sending the right code back up the calling sequence.

When code returns the "noproblem" value, all callers will complete what they are doing, but if user output was created will likely not initiate any new rules.

Functions

Topics are not functions and do not take arguments. CS provides system functions and you can write user functions in ChatScript.

Function names always start with ^, like `^match(argument1 argument2)` and **no commas are used to separate the arguments** (since commas themselves might be legal arguments).

These are classic functions in that they have arguments and a collection of code to execute. Their code can generate output and/or make calls to other functions, including invoking topics and rules. Functions are a convenient way to abstract and share code.

Call by value

```
outputmacro: ^myfunction($_argument1 $_argument2)
    $_argument1 += 1
```

Use of `$_` variables in the function definition is a call by value.

All `$_` variables are purely local and cannot be seen outside of the function (or topic) they are used in. This is the preferred way to call, unless you need to write back to your caller.

Call by reference

ChatScript also has function argument variables, whose names always start with ^ and have local (lexical) visibility but implement call by reference. You can assign back to the caller and write onto the variable he passed you.

For outputmacros:

```
   outputmacro: ^myfunction(^argument1 ^argument2)
        ^argument1 += 1
```

However, unless you need call by reference (being able to assign to the variable and have it affect the caller) you should use call by value so that nothing outside your routine can impact it.

Patternmacros, however, do not normally ever write onto their arguments, so it is not only safe to use function arguments `^argument1`, but necessary since patternmacros are not really functions at all. They merely temporarily paste their code into the pattern stream and so do not save and restore variable values or have locals per se.

```
patternmacro: ^myfunction(^argument1 ^argument2)
```

You can mix call by reference and call by value arguments.

An alternate function format allows you to put the output code within {}, which is more nicely visualized by some editors.

```
outputmacro: ^myfunction(^argument1 ^argument2)
{
^argument1 += 1
```

```
}
```

Whenever you see a function variable, you can imagine it is as though the script had its argument immediately substituted in. This is a call by reference. So if the script call was this

```
^myfunction($myvar 1)
```

then the effect of `^argument1 += 1` is as though `$myvar += 1` were done and `$myvar` would now be one higher.

Of course, had you tried to do `^argument2 += 1` then that would be the illegal `1 += 1` and the assignment would fail.

ADVANCED TOKENIZATION

The CS natural language workflow consists of taking the user's input text, splitting it into tokens and stopping each time at a perceived sentence boundary. It continues with the input after processing that "sentence". That leaves two tricky bits: what is a token and what is a sentence boundary. The '$cs_token~ variable gives you some control over how these work. The naive definition of a token is a sequence of letters terminating in a space or end of input. But there are exceptions to that like some kind of sentence punctuation (comma, period, colon, exclamation) is not part of a bigger token. The sentence punctuation notion has exceptions, like the period within a floating point number or as part of an abbrviation or webaddress. And hyphens with more letters on the other side are generally not punctuation either. And normally we consider bracketing things like parens not part of a word (except in emoticons). So CS will normally break things apart as it believes they should be done. If you need to actually allow a token to have embedded punctuation in it, you can list the token in the LIVEDATA/SUBSTITUTES/abbreviations.txt file and the tokenizer will respect it.

System Functions

There are many system functions to perform specific tasks. These are enumerated in the ChatScript System Functions Manual and the ChatScript Fact Manual.

Out of band Communication

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

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

```
[ speed=10 rate: 50 ] User said this
```

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

Here is a sample pattern to catch oob data.

```
u: ( < \[ * speed _*1 * \] ) The speed is _0

u: ( < \[ * rate _*1 * \] ) The rate is _0
```

You need `*` in front of your data when you can have multiple forms of data and you need `* \]` after your data to ensure you don't match words from user input.

On output you need to do one of these

```
u: () \[ oob data \] Here is user message

u: () ^"[oob data] Here is user message
```

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

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

You can hand author gestures directly on your outputs, but then you have to be certain you only output one sentence at a time from your chatbot (lest a gesture command get sandwiched between two output sentence). You also have to be willing to hand author the use of each gesture.

I prefer to write patterns for common things (like shake head no or nod yes) and have the system automatically generate gestures during postprocessing on its own output.

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

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

OOB Output	description
Loopback	The webpage or stand-alone engine will wait for the designated milliseconds after every output from CS and if the user has not begun typing will send in the oob message [loopback] to CS. If user begins typing before the timeout, the loopback is cancelled for this output only, and will resume counting on the next output. e.g. `[loopback=3000]` will wait 3 seconds after every output.
Alarm	The webpage or stand-alone engine will wait for the designated milliseconds and then send in the oob message [alarm] to CS. Input typing has no effect. e.g. `[alarm=3000]` will wait 3 seconds and then send in the alarm. CS can cancel any of these by sending an oob message with a milliseconds of 0. e.g. `[loopback=0 callback=0 alarm=0]` cancels any pending callbacks into the future.

System callback functions

`^CSBOOT()`

`outputmacro: ^CSBOOT()`

This function, if defined by you, will be executed on startup of the ChatScript system. It is a way to dynamically add facts and user variables into the base system common to all users (the boot layer). And returned output will go to the console and if a server, into the server log Note that when a user alters a system `$variable`, it will be refreshed back to its original value for each user.

If you create JSON data, you should probably use ^jsonlabel() to create unique names separate from the normal json naming space.

You can also add facts to the boot layer by creating facts from a user script using ^createFact(s v o #FACTBOOT)

This function allows you to read external data to be incorporated as Facts into the basic server. You are allowed to define any number of these in a multibot environment without triggering a complaint about already defined functions.

^purgeboot can be used to erase facts stored in the boot layer.

^CS_REBOOT()

`outputmacro: ^CS_REBOOT()`

This function, if defined by you, will be executed on every volley prior to loading user data. It is executed as a user-level program which means when it has completed all newly created facts and variables just disappear. It is used in conjunction with a call to the system function ^reboot() to replace data from a ^CSBOOT. Typically you would have the ^CS_REBOOT function test some condition (results of a version stamp) and if the version currently loaded in the boot layer is current, it simply returns having done nothing. If the boot layer is not current, then you call ^REBOOT() which erases the current boot data and treats the remainder of the script as refilling the boot layer with new facts and variables.

^CSSHUTDOWN()

`outputmacro: ^CSSHUTDOWN()`

This function, if defined by you, will be executed on shutdown or restart of the ChatScript system.

^cs_topic_enter()

`outputmacro: ^cs_topic_enter(^topic ^mode)`

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

^cs_topic_exit()

`outputmacro: ^cs_topic_exit(^topic ^result)`

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

AutoInitFile

When a user is initialized for the first time, the system will attempt to read a top-level file named for the user as `bruce-init.txt` (if user is bruce). If found, commands will be executed from there (analogous to the `:source` command. This will be read after any `source=` command line parameter.

Advanced :build

Anti-virus software and :build

Windows Defender, Norton, and the like have a real-time monitoring system on files. You can disable the ChatScript folder from being analyzed. On a Mac w/o this stuff, a compile of a bot might take 14 seconds, wherease with AV software interferring on Windows it takes 4 minutes. CS writes to its TOPIC folder and LOGS directories in lots of little pieces, that AV wants to monitor.

:build xxx quiet

Build normally echos out its log messages of what it is currently compiling And and any warning or error messages. If you say quiet, then it will only tell you it succcessfully completed or list the errors it detected.

Build warning messages

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

```
:build 0 nospell
```

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

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

```
:build 0 outputspell
```

run spellchecking on text output of rules (see if typos exist).

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

```
topic: ~mytopic ( cheese !cheese)
```

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

```
topic: ~mytopic (cheese cheese~1)
```

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

```
topic: ~mytopic (cheese cheese~n)
```

Since you have accepted all forms of cheese, you don't need to name `cheese~n`. `:build` also warns you about various substitutions that might affect your patterns. You can suppress those messages with `:build filename nosubstitution`

Files

When you name a file or directory, :build will ignore files that do not end in .top or .tbl . When you name a directory, it walks all the files in that directory, but does not recurse into subdirectories unless you explicitly ask it to by adding a second slash after the directory name. If the contents of your filesxxx build file had this:

```
topic.top
subdirectory1/
subdirectory2//
```

then it would compile topic.top, all files within subdirectory1 non-recursively, and all files recursively in subdirectory2.

Trace

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

```
:build harry trace
```

Reset User-defined

Normally, a build will leave your current user identity alone. All state remains unchanged, except that topics you have changed will be reset for the bot (as though it has not yet ever seen those topics). But if you want to start over with the new system as a new user, you can request this on the build command.

```
:build 0 reset
```

reinit the current user from scratch (equivalent to `:reset user`).

Build Layers

The build system has two layers, 0 and 1. When you say :build 0, the system looks in the top level directory for a file `files0.txt`. Similarly when you say `:build 1` it looks for `files1.txt`. Whatever files are listed inside a `filesxxx.txt` are what gets built.

And the last character of the file name (e.g., `files0`) is what is critical for deciding what level to build on. If the name ends in 0, it builds level 0. If it doesn't, it builds level 1. This means you can create a bunch of files to build things any way you want. You can imagine:

- `:build common0` - shared data-source (level 0)
- `:build george` - george bot-specific (level 1)
- `:build henry` - henry bot-specific (level 1)
- `:build all` - does george and henry and others (level 1)
- `:build system0` - does ALL files, there is no level 1.

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

Note

Avoid something likes `files2.txt` and doing a `:build 2`. 2 specifies a level and normal bots are at level 1 (which requires no numbering). Name your file after your bot and it will default to level 1.

Skipping a topic file

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

Block comments

Normally `#` becomes a comment to end of line. But you can use a block comment as follows:

```
##<< first junk
some junk
##>> more junk
```

Because any comment marker kills the rest of the line, the "first junk" will not be seen, nor will the "more junk". But a comment block was established, so lines between them line "some junk" are also not seen.

Renaming Variables, Sets, and Integer Constants

A top level declaration in a script can rename a match variable

```
rename: _bettername _12
```

before any uses of `_bettername`, which now mean `_12`. You can put multiple rename pairs in the same declaration.

```
rename: _bettername _12 _okname _14
```

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

```
rename: _xname _12
```

and both _xname and `_bettername` refer to `_12`.

Renames can also rename concept sets:

```
rename: @myset @1
```

so you can do:

```
@myset += createfact( 1 2 3)
$$tmp = first(@mysetsubject)
```

You can also declare your own 32 or 64-bit integer constants. You must use ## when you define it and when you refer to it.

```
rename: ##first 1
$tmp = ##first
```

Defining private Queries

see ChatScript Fact Manual.

Documenting variables, functions, factsets, and match variables

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

```
describe: $myvar "used to store data"
 _10 "tracks pos tag"
```

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

Conditional compilation

You can have the system include or exclude lines on a line by line basis. To make a line conditional, put a comment left justified where a word is contiguous to the #, like this:

```
#german u: (test) this is conditionally compiled
```

This line is normally ignored because it is a comment line and not a named numeric constant. But if you put the `#german` as a tail parameter of the `:build` command, you enable it:

You can also handle blocks of code analogous to the block comment convention by appending a label to the <<## :

```
<<##german ...
... >>##
```

```
:build Harry #german
```

You may name up to 9 conditions on your build line. In fact, for language-related conditional lines, you don't have to declare anything on the `:build` command. The system will automatically accept lines that name the current language= command line parameter (English being the default).

Conditional compilation applies to script files and the filesxxx.txt files and LIVEDATA files.

A Fresh Build

You've been building and chatting and something isn't right but it's all confusing. Maybe you need a fresh build. Here is how to get a clean start.

- Quit chatscript.
- Empty the contents of your USER folder, but don't erase the folder. This gets rid of old history in case you are having issues around things you've said before or used from the chatbot before.
- Empty the contents of your TOPIC folder, but don't erase the folder. This gets rid of any funny state of topic builds.

`:build 0` - rebuild the common layer `:build xxx` - whatever file you use for your personality layer

Probably all is good now. If not quit chatscript. Start up and try it now.

Editing Non-topic Files

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

DICT files

You may choose to edit the dictionary files. There are 3 kinds of files.

The `facts0.txt` file contains hierarchy relationships in wordnet. You are unlikely to edit these.

The `dict.bin` file is a compressed dictionary which is faster to read. If you edit the actual dictionary word files, then erase this file. It will regenerate anew when you run the system again, revised per your changes. The actual dictionary files themselves... you might add a word or alter the type data of a word. The type information is all in `dictionarySystem.h`

LIVEDATA files

The substitutions files consistof pairs of data per line. The first is what to match. Individual words are separated by underscores, and you can request sentence boundaries `<` and `>` .

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

file	description
`interjections.txt`	remaps to `~` words standing for interjections or discourse acts
`contractions.txt`	remaps contractions to full formatting
`substitutes.txt`	(omittable) remaps idioms to other phrases or deletes them.
`british.txt`	(omittable) converts british spelling to us
`spellfix.txt`	(omittable) converts a bunch of common misspellings to correct
`texting.txt`	(omittable) converts common texting into normal english.
`systemessentials.txt`	things needed to handle end punctuation
`expandabbreviations.txt`	does what its name suggests
`queries.txt`	defines queries available to `^query`. A query is itself a script. See the file for more information.

file	description
`canonical.txt`	is a list of words and override canonical values. When the word on the left is seen in raw input, the word on the right will be used as its canonical form.
`lowercasetitles.txt`	is a list of lower-case words that can be accepted in a title. Normally lower case words would break up a title.

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

Common Script Idioms

Selecting Specific Cases `^refine`

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

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

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

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

topic: ~subcountry system[]

u: (Turkey) ...
u: (Sweden) ...
u: (*) ...
```

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

```
?: (<what) ^respond(~quibblewhat)
?: (<when) ^respond(~quibblewhen)
?: (<who) ^respond(~quibblewho)

# ...
```

```
topic: ~quibblewho system []

?: ( <who knows ) The shadow knows
?: ( <who can ) I certainly can't.
```

Using ^reuse

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

```
t: HOUSE () I live in a small house

u: ( where * you * live ) ^reuse(HOUSE)
```

The rule on disabling a rule after use is that the rule that actually generates the output gets disabled. So the default behavior (if you don't set keep on the topic or the rule) is that if the question is asked first, it reuses HOUSE.

Since we have given the answer, we don't want to repetitiously volunteer it, HOUSE gets disabled. But, if the user repetitiously asks the question (maybe he forgot the answer), we will answer it again because the responder didn't get disabled, just the gambit. And disabling applies to allowing a rule to try to match, not to what it does for output. So one can reuse that gambit's output any number of times.

If you don't want that behavior you can either add a disable on the responder OR tell ^reuse to skip used rules by giving it a second argument (anything). So one way is:

```
t: HOUSE () I live in a small house

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

and the other way is:

```
t: HOUSE () I live in a small house

u: ( where * you * live ) ^reuse(HOUSE skip)
```

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

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

```
t: HOUSE () I live in a small house $house = 1
```

```
u: ( where * you * live ) [$house You forgot?] ^reuse(HOUSE)
```

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

```
outputmacro: ^heforgot(^test) [^test You forgot?]
    t: HOUSE () I live in a small house $house = 1

    u: ( where * you * live ) heforgot($house ) ^reuse(HOUSE)
```

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

```
outputmacro: ^heforgot(^test) [^test You forgot?] ^test = 1
    t: HOUSE () heforgot($house ) I live in a small house.

    u: ( where * you * live ) ^reuse(HOUSE)
```

Esoterica and Fine Detail

Being first to converse

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

Prefix labeling in stand-alone mode

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

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

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

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

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

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

How are double-quoted strings handled?

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

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

I liked "War and Peace"

It also happens as keywords in concepts:

```
concept: ~test[ "kick over"]
and in tables:
DATA:
"Paris, France"
```

and in patterns:

```
u: ( "do you know" what )
```

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

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

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

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

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

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

What really happens on the output side of a rule?

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

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

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

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

```
u: (x)
$var2 = apples
$var1= join($ var2)
I like $var1
```

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

This evaluation during output is in contrast to the behavior on the pattern side where the goal is presence, absence, and failure. Naming a word means finding it in the sentence.

Naming a concept/topic means finding a word which inherits from that concept either directly or indirectly. Naming a variable means seeing if that variable has a non-null value.

Calling a function discards any output stream generated and aside from other side effects means did the function fail (return a fail code) or not.

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

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

```
t: I like (somewhat) ice
```

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

```
t: I fail (sort of) at most things
```

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

How should I go about creating a responder?

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

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

What really happens with rule erasure?

The system's default behavior is to erase rules that put output into the output stream, so they won't repeat themselves later. You can explicitly make a rule erase with `^disable()` and not erase with `^keep()` and you can make the topic not allow responders to erase with keep as a topic flag.

So, if a rule generates output, it will try to erase itself. If a rule uses `^reuse()`, then the rule that actually generated the output will be the called rule. If for some reason it cannot erase itself, then the erasure will rebound to the caller, who will try to erase himself.

Similarly, if a rule uses `^refine()`, the actual output will come from a `rejoinder()`. These can never erase themselves directly, so the erasure will again rebound to the caller.

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

`u: (~emogoodbye)`

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

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

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

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

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

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

Control Flow

There is no GOTO in chatscript. There are only calls. Calls always return. They return with noproblem or end or fail.

Respond/Gambit calls enter a new topic. Any end/fail TOPIC will terminate them and return to the caller.

`end(TOPIC)` has no consequence to the caller.

`fail(TOPIC)` degrades to a fail-rule and terminates the calling rule unless the call was wrapped in NOFAIL().

`Nofail(TOPIC)` and `Nofail(RULE)` are equivalent (because you can't get back a failed topic flag). A call to reuse does not enter a new topic context, so if the reuse calls `end(topic)`, that returns to the calling rule, which has received an end(topic). If not wrapped in a `nofail(TOPIC)`, then the calling rule terminates with end topic.

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

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

Pattern Matching Anomolies

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

```
u: (* here *) xxx
```

you will succeed. The actual tokens are "we" "go" "here and there" "about" "town". but the pattern matcher is allowed to peek some into composite words.

When it does match, since the actual token is *"here and there"*, the position start is set to that word (e.g., position 3), and in order to allow to match other words later in the composite, the position end is set to the word before (e.g., position 2). This means if your pattern is

```
u: (* here and there *) xxx
```

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

```
u: (_* and _* ) xxx
```

then `_0` is bound to words 1 & 2 "we go", and matches "here and there", and `_1` matches the rest, "about town".

That is, the system will NOT position the position end before the composite word. If it did, `_1` would be *here and there about town*. It's not.

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

```
u: ( * _and * ) xxx
```

would memorize the empty word for `_0`. If you don't want something within a word to match your word, you can always quote it.

```
u: ( X * ‘and * ) xxx
```

does not match *here and there about town*.

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

```
concept: ~myjunk (and)
u: ( * _~myjunk * ) xxx
```

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

```
u: ( * _'~myjunk * ) xxx
```

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

```
concept: ~myfamily (brother)
u: ( * _'~ myfamily * ) xxx
```

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

```
concept: ~myfamily (‘brother)
```

Blocking a topic from accidental access

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

```
t: (!~) ^fail(topic)
```

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

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

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

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

Self-Reflection

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

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

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

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

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

Reflective information is available during main processing as well. You can set `%why` to be a value and that value will be associated with any output generated thereafter. E.g., `%why = quibble`. The system also sets `$cs_tokencontrol` to results that happen from input processing.

Updating CS Versions Easily

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

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

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

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

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

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

The Dictionary

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

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

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

[Wiki home] - [Basic User Manual]