ChatScript Advanced User's Manual

© Bruce Wilcox, gowilcox@gmail.com www.brilligunderstanding.com Revision 8/23/2018 cs8.5

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 onto 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.

ADVANCED CONCEPTS

Concepts can have part of speech information attached to them (using `dictionarysystem.h` values). Eg.

```
concept: ~mynouns NOUN NOUN_SINGULAR (boxdead foxtrot)
concept: ~myadjectives ADJECTIVE ADJECTIVE_BASIC (moony dizcious)
```

Since the script compile issues warning messages on words it doesn't recognize, in case you misspelled them, you can also add `IGNORESPELLING` as a flag on the concept:

```
concept: ~unknownwords IGNORESPELLING (asl daghh)
```

and you can combine pos declarations and ignorespelling. This is applied recursively to any concepts that are members of this concept. That may be a bit excessive.

Rather than assigning parts of speech you can recursively limit a concept's words to a part of speech using `ONLY_NOUNS`, `ONLY_VERBS`, `ONLY_ADJECTIVES`, or `ONLY_ADVERBS`.

```
Concept: ~verbs ONLY_VERBS (sit sleep)
```

This will not react to noun meanings of sleep. The current ontology files for verbs, adverbs, and adjectives all have the appropriate `ONLY` marked on them.

When you don't want a member concept marked as a consequence, you can use `ONLY_NONE` to block propogate. Thus:

```
concept: ~verbs ONLY_VERBS (~active_verbs sit)
concept: ~active_verbs ONLY_NONE (sleep)
```

will prevent sleep from being required to be a verb form. Note that verb forms do not include verbs used as nouns (ie gerunds).

Normally if you declare a concept a second time, the system considers that an error. If you add the marker `MORE` to its definition, it will allow you to augment an existing list.

```
concept: ~city MORE (Tokyo)
```

Normally concepts (and topics) discard repeated keywords. For concepts, you can force it to allow repeats using `DUPLICATE`

```
concept: ~mapword DUPLICATE (year month day year month day) # the concept has 6 members
```

Concepts can be built from other concepts that do not have specific words.

```
Concept: ~myconcept (!thisword ~otherconcept)
```

Note: the system has two kinds of concepts.

- *Enumerated* concepts are ones formed from an explicit list of members. Stuff in definitions of `concept: ~xxx()` are that.
- There are also *internal* concepts marked by the system. These include part of speech of a word (requires using the pos-tagger to decide from the input what part of speech it was of possibly several), grammatical roles, words from infinite sets like `~number` and `~placenumber` and `~weburl`, and so forth.

In a pattern of some kind, if you are referencing a sentence location using a match variable, you can match both kinds of concepts. But if you are not tied to a location in a sentence, you can't match internally computed ones. So something like

```
if ( pattern 23?~number )
```

will fail. Even

```
if ( pattern practical?~adjective )
```

will fail given that deciding practical is an adjective (it could also be a noun) hasn't been performed by pos-tagging.

All internal concepts are members of the concept `~internal_concepts`.

ADVANCED TOPICS

There are several things to know about advanced topics.

Topic Execution

When a topic is executing rules, it does not stop just because a rule matches. It will keep executing rules until some rule generates ouput for the user or something issues an appropriate `^end` or `^fail` call. So you can do things like this:

```
u: ( I love ) $userloves = true

u: ( dog ) $animal = dog

u: ( love ) Glad to hear it

u: ( dog ) I hate dogs
```

and given *I love dogs*, the system will set $userloves and $animal and output *glad to hear it*.

Topic Control Flags

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

```
topic: ~rust keep random [rust iron oxide]
```

The flags and their meanings are:

flag	description
`random`	search rules randomly instead of linearly

flag	description
`norandom`	(default) search rules linearly
`keep`	do not erase responders ever. Gambits (and rejoinders) are not affected by this
`erase`	(default) erase responders that successfully generate output.Gambits automatically erase unless you suppress them specifically.
`nostay`	do not consider this a topic to remain in, leave it (except for rejoinders)
`stay`	(default) make this a pending topic when it generates output
`repeat`	allow rules to generate output which has been output recently
`norepeat`	(default) do not generate output if it matches output made recently
`priority`	raise the priority of this topic when matching keywords

flag	description
`normal`	(default) give this topic normal priority when matching keywords
`deprioritize`	lower the priority of this topic when matching keywords
`system`	this is a system topic. It is automatically `nostay`, `keep`.`keep` automatically applies to gambits as well. The system never looks to these topics for gambits. System topics can never be considered pending (defined shortly). They can not have themselves or their rules be enabled or disabled. Their status/data is never saved to user files.
`user`	(default) this is a normal topic
`noblocking`	should not perform any blocking tests on this topic in `:verify`

flag	description
`nopatterns`	should not perform any pattern tests on this topic in `:verify`
`nosamples`	should not perform any sample tests on this topic in `:verify`
`nokeys`	should not perform any keyword tests on this topic in `:verify`
`more`	normally if you try to redeclare a concept, you get an error. `more` tells CS you intend to extend the concept and allows additional keywords.
`bot=name`	if this is given, only named bots are allowed to use this topic. See ChatScript Multiple Bots manual.

Rules that erase and repeat

Normally a rule that successfully generates output directly erases itself so it won't run again. Gambits do this and responders do this.

Gambits will erase themselves even if they don't generate output. They are intended to tell a story or progress some action, and so do their thing and then disappear automatically.

Rejoinders don't erase individually, they disappear when the rule they are controlled by disappears. A rule that is marked keep will not erase itself. Nor will responders in a topic marked keep (but gambits still will).

Responders that generate output erase themselves. Responders that cause others to generate output will not normally erase themselves (unless...):

```
u: ( * ) respond(~reactor)
```

If the above rule causes output to be generated, this rule won't erase itself, the rule invoked from the `~reactor` topic that actually generated the output will erase itself. But, if the rule generating the output is marked keep, then since someone has to pay the price for output, it will be this calling rule instead.

`Repeat` does not stop a rule from firing, it merely suppresses its output. So the rule fires, does any other effects it might have, but does not generate output. For a responder, if it doesn't generate output, then it won't erase itself. For a gambit, it will because gambits erase themselves regardless of whether they generate output or not.

Keywords vs Control Script

A topic can be invoked as a result of its keywords or by a direct call from the control script or some other topic. If you intend to call it from script, then there is almost never any reason to give it keywords as well, because that may result in it being called twice, which is wasteful, or out of order, if there was a reason for the point you called it from script.

Pending Topics

The second thing to know about topics is what makes a topic pending. Control flow passes through various topics, some of which become pending, meaning one wants to continue in those topics when talking to the user. Topics that can never be pending are: system topics, blocked topics (you can block a topic so it won't execute), and nostay topics.

What makes a remaining topic pending is one of two things. Either the system is currently executing rules in the topic or the system previously generated a user response from the topic. When the system leaves a topic that didn't say anything to the user, it is no longer pending. But once a topic has said something, the system expects to continue in that topic or resume that topic.

The system has an ordered list of pending topics. The order is:

- 1st- being within that topic executing rules now,
- 2nd- the most recently added topic (or revived topic) is the most pending.

You can get the name of the current most pending topic(`%topic`), add pending topics yourself (`^addtopic()`), and remove a topic off the list (`^poptopic()`).

Random Gambit

The third thing about topics is that they introduce another type, the random gambit, `r:`.

The topic gambit `t:` executes in sequence forming in effect one big story for the duration of the topic. You can force them to be dished randomly by setting the random flag on the topic, but that will also randomize the responders. And sometimes what you want is semirandomness in gambits.

That is, a topic treated as a collection of subtopics for gambit purposes. This is `r:` The engine selects an `r:` gambit randomly, but any `t:` topic gambits that follow it up until the next random gambit are considered "attached" to it. They will be executed in sequence until they are used up, after which the next random gambit is selected.

```
Topic: ~beach [beach sand ocean sand_castle]

# subtopic about swimming
r: Do you like the ocean?

t: I like swimming in the ocean.

t: I often go to the beach to swim.

# subtopic about sand castles.
r: Have you made sand castles?
    a: (~yes) Maybe sometime you can make some that I can go see.
    a: (~no) I admire those who make luxury sand castles.

t: I've seen pictures of some really grand sand castles.
```

This topic has a subtopic on swimming and one on sand castles. It will select the subtopic randomly, then over time exhaust it before moving onto the other subtopic.

Note any `t:` gambits occurring before the first `r:` gambit, will get executed linearly until the `r:` gambits can fire.

Overview of the control script

Normally you start using the system with the pre-given control script. But it's just a topic and you can modify it or write your own.

The typical flow of control is for the control script to try to invoke a pending rejoinder. This allows the system to directly test rules related to its last output, rules that anticipate how the user will respond.

Unlike responders and gambits, the engine will keep trying rejoinders below a rule until the pattern of one matches and the output doesn't fail.

Not failing does not require that it generate user output. Merely that it doesn't return a fail code. Whereas responders and gambits are tried until user output is generated (or you run out of them in a topic).

If no output is generated from rejoinders, the system would test responders. First in the current topic, to see if the current topic can be continued directly. If that fails to generate output, the system would check other topics whose keywords match the input to see if they have responders that match. If that fails, the system would call topics explicitly named which do not involve keywords. These are generic topics you might have set up.

If finding a responder fails, the system would try to issue a gambit. First, from a topic with matching keywords. If that fails, the system would try to issue a gambit from the current topic. If that fails, the system would generate a random gambit.

Once you find an output, the work of the system is nominally done. It records what rule generated the output, so it can see rejoinders attached to it on next input. And it records the current topic, so that will be biased for responding to the next input. And then the system is done. The next input starts the process of trying to find appropriate rules anew.

There are actually three control scripts (or one invoked multiple ways). The first is the preprocess, called before any user sentences are analyzed. The main script is invoked for each input sentence. The postprocess is invoked after all user input is complete. It allows you to examine what was generated (but not to generate new output except using special routines `^postprintbefore` and `^postprintafter`).

ADVANCED PATTERNS

Keyword Phrases

You cannot make a concept out with a member whose string includes starting or trailing blanks, like " X ". Such a word could never match as a pattern, since spaces are skipped over. But you can make it respond to idiomatic phrases and multiple words. Just put them in quotes. E.g.

```
concept: ~remove ( "take away" remove )
```

Normally in patterns you can write

```
?: ( do you take away cheese )
```

and the system will match sentences with those words in order.

In WordNet, some words are actually composite words like : *TV_show*. When you do `:prepare` on *what is your favorite TV show* you will discover that the engine has merged *TV_show* into one composite word. The system has no trouble matching inputs where the words are split apart

```
?: ( what is your favorite TV show )
```

But if you tried a word memorize like

```
?: ( what is your favorite *1 *1 )
```

that would fail because the first `*1` memorizes *TV_show* and there is therefore no second word to memorize.

Likewise when you write

```
concept: ~viewing ("TV show")`
```

the system can match that concept readily also. In fact, whenever you write the quoted keyword phrase, if all its words are canonical, you can match canonical and noncanonincal forms. *"TV show"* matchs *TV shows* as well as *TV show*.

Implied concept Sets

When you make a pattern using [] or {} and it only contains words, phrases, And concept sets, the system will make an anonymous concept set out of them. This allows the system to find the soonest match of any of them. otherwise [] and {} take each element in turn and try to find a match, which may be later in the sentence than a later element in the set would match.

Dictionary Keyword sets

In ChatScript, WordNet ontologies are invoked by naming the word, a `~`, and the index of the meaning you want.

```
concept: ~buildings [ shelter~1 living_accomodations~1 building~3 ]
```

The concept `~buildings` represents 760 general and specific building words found in the WordNet dictionary – any word which is a child of: definition 1 of shelter, definition 1 of accommodations, or definition 3 of building in WordNet's ontology.

How would you be able to figure out creating this? This is described under `:up` in Word Commands later.

`Building~3` and `building~3n` are equivalent.

The first is what you might say to refer to the 3rd meaning of building. Internally `building~3n` denotes the 3rd meaning and its a *noun* meaning.

You may see that in printouts from Chatscript. If you write 3n yourself, the system will strip off the `n` marker as superfluous.

Similarly you can invoke parts of speech classes on words. By default you get all of them. If you write:

```
concept: ~beings [snake mother]
```

then a sentence like *I like mothering my baby* would trigger this concept, as would *He snaked his way through the grass.* But the engine has a dictionary and a part-of-speech tagger, so it often knows what part of speech a word in the sentence is.

You can use that to help prevent false matches to concepts by adding `~n ~v ~a` or `~b` (adverb) after a word.

```
concept: ~beings [snake~n mother~n]
```

If the system isn't sure something is only a noun, it would let the verb match still. Thus a user single-word reply of snakes would be considered both *noun* and *verb.*

The notation `run~46` exists to represent a meaning.

There is mild inherent danger that I might kill off some word meaning that is problematic (eg if `run~23` turned out mark the `~curses` set and I didn't want the resulting confusion), said kill off might strand your meaning by renumbering into either non-existence (in which case the script compiler will warn you) or into a different pos set (because your meaning was on the boundary of meanings of a different pos type).

Use of the specific meaning is handy in defining concepts when the meaning is a noun, because Wordnet has a good noun ontology.

Use of the specific meaning of other parts of speech is more questionable, as Wordnet does not have much ontology for them.

The broader scope meaning restriction by part-of-speech (eg `run~v`) has much more utility. It has its risks in that it depends on the parser getting it right (as you have seen), which over time will get better and better.

In MOST cases, you are better off with the full fledged unadorned word, which is parse-independent.

This is particularly true when you are pattern matching adjacent words and so context is firm. < run `~app` is a pretty clean context which does not need pos-certification.

The topic on `~drugs` would want in its keyword list `clean~a` to allow *I've been clean for months* to target it, but not *I clean my house.*

System Functions

You can call any predefined system function. It will fail the pattern if it returns any fail or end code. It will pass otherwise. The most likely functions you would call would be:

`^query` – to see if some fact data could be found. Many functions make no sense to call, because they are only useful on output and their behavior on the pattern side is unpredictable.

Macros

Just as you can use sets to "share" data across rules, you can also write macros to share code.

Pattern macros

A `patternmacro` is a top-level declaration that declares a name, arguments that can be passed, and a set of script to be executed "as though the script code were in place of the original call".

Macro names can be ordinary names or have a `^` in front of them. The arguments must always begin with `^`.

The definition ends with the start of a new top-level declaration or end of file. E.g.

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

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

The above patternmacro takes one argument (who we are talking about). After checking that the sentence is not in the negative, it uses a choice to consider alternative ways of asking what the hair color is.

The first way matches are you a redhead. The second way matches what is my hair color. The call passes in the value *I* (which will also match my mine etc in the canonical form).

Every place in the macro code where `^who` exists, the actual value passed through will be used.

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

Whereas most programming language separate their arguments with commas because they are reserved tokens in their language, in ChatScript a comma is a normal word. So you separate arguments to functions just with spaces.

```
?: ( ^FiveArgFunction(1 3 my , word) )
```

When a patternmacro takes a single argument and you want to pass in several, you can wrap them in parens to make them a single argument. Or sometimes brackets. E.g.,

```
?: ( ^DoYouDoThis( (play * baseball) ) ) Yes I do

?: ( ^DoYouDoThis( [swim surf "scuba dive"] ) Yes I do
```

If you call a patternmacro with a string argument, like *"scuba dive"* above, the system will convert that to its internal single-token format just as it would have had it been part of a normal pattern. Quoted strings to output macros are treated differently and left in string form when passed.

You can declare a patternmacro to accept a variable number of arguments. You define the macro with the maximum and then put "variable" before the argument list. All missing arguments will be set to null on the call.

```
patternmacro: ^myfn variable (^arg1 ^arg2 ^arg3 ^arg4)
```

Dual macros

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

Patternmacro cannot be passed a factset name. These are not legal calls:

```
^mymacro(@0)
^mymacro(@0subject)
```

Do not write code in a pattern macro as though it is an output code. You can't do

```
if (...) {}
```

If you want to do that, use an outputmacro and call that from your pattern.

Literal Next \

If you need to test a character that is normally reserved, like `(` or `[`, you can put a backslash in front of it.

```
s: ( \( * \) ) Why are you saying that aside?
```

This also works with entire tokens like:

```
u: ( \test=fort )
```

Normally the above without \ would be considered a comparison. But the \ at the start of it says treat = as just an ordinary part of the token. You can even put _ in front of it:

```
u: ( _\test=fort )
```

Note that \ does not block a word with an * in it from performing wildcard spelling.

Question and exclamation - ? !

Normally you already know that an input was a question because you used the rule type ?: .

But rejoinders do not have rule types, so if you want to know if something was a question or not, you need to use the ? keyword. It doesn't change the match position

```
t: Do you like germs?
    a: ( ? ) Why are you asking a question instead of answering me?
    a: ( !? ) I appreciate your statement.
```

If you want to know if an exclamation ended his sentence, just backslash a ! so it won't be treated as a not request. This doesn't change the match position.

```
s: ( I like \! ) Why so much excitement
```

More comparison tests - & ?

You can use the logical and bit-relation to test numbers. Any non-zero value passes.

```
s: ( _~number _0&1 ) Your number is odd.
```

? can be used in two ways. As a comparison operator, it allows you to see if the item on the left side is a member of a set (or JSON array) on the right. E.g.

```
u: ( _~propername '_0?~bands )
```

As a standalone, it allows you to ask if a wildcard or variable is in the sentence. E.g.

```
u: ( _1? )
u: ( $bot? )
```

Note that when `_1` is a normal word, that is simple for CS to handle. If `_1` is a phrase, then generally CS cannot match it. This is because for phrases, CS needs to know in advance that a phrase can be matched.

If you put *take a seat* as a keyword in a concept or topic or pattern, that phrase is stored in the dictionary and marked as a pattern phrase, meaning if the phrase is ever seen in a sentence, it should be noticed and marked so it can be matched in a pattern. But if it is merely in a variable, then the dictionary is unaware of the phrase and so `_1?` will not work for it.

Comparison with C++ #define in `dictionarysystem.h`

You can name a constant from that file as the right hand side of a comparison test by prefixing its name with `#`. E.g.,

```
s: ( _~number _0=#NOUN )
```

Such constants can be done anywhere, not just in a pattern.

Current Topic ~

Whenever you successfully execute a rule in a topic, that topic becomes a pending topic (if the topic is not declared system or nostay). When you execute a rule, while the rule is obviously in a topic being tested, it is not necessarily a topic that was in use recently.

You can ask if the system is currently in a topic (meaning was there last volley) via `~`. if the topic is currently on the pending list, then the system will match the `~`. E.g.,

```
u: ( chocolate ~ ) I love chocolate ice cream.
```

The above does not match any use of chocolate, only one where we are already in this topic (like topic: `~ice_cream`) or we were recently in this topic and are reconsidering it now.

A useful idiom is `[~topicname ~]`. This allows you to match if EITHER the user gave keywords of the topic OR you are already in the topic. So:

```
u: ( << chocolate [~ice_cream ~] >> )
```

would match if you only said *chocolate* while inside the topic, or if you said *chocolate ice cream* while outside the topic.

Prefix Wildcard Spelling and Postfix Wildcard Spelling

Some words you just know people will get wrong, like *Schrodinger's cat* or *Sagittarius*.

You can request a partial match by using an * to represent any number of letters (including 0). For example

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

u: ( *tor ) # this matches "reactor".
```

The * can occur in one or more positions and means 0 or more letters match.

A period can be used to match a single letter (but may not start a wildcardprefix). E.g.,

```
u: ( p.t* )
```

can match pituitary or merely pit or pat. You cannot use a wildcard on the first letter of the pattern.

```
u: ( .p* )
```

is not legal because it may not start with a period.

Indirect pattern elements

Most patterns are easy to understand because what words they look at is usually staring you in the face.

With indirection, you can pass patterndata from other topics, at a cost of obscurity. Declaring a macro does this. A ^ normally means a macro call (if what follows it is arguments in parens), or a macro argument. The contents of the macro argument are used in the pattern in its place. Macro arguments only exist inside of macros. But macros don't let you write rules, only pieces of rules.

Normally you use a variable directly. `$$tmp = nil` clears the `$$tmp` variable, for instance, while `u: ( ) $$tmp` goes home will output the value into the output stream. The functional user argument lets you pass pattern data from one topic to another.

```
s: ( are you a _^$var )
```

The contents of `$var` are used at that point in the pattern. Maybe it is a set being named. Maybe it's a word. You can also do whole expressions, but if you do you will be at risk because you won't have the script compiler protecting you and properly formatting your data. See also Advanced Variables.

Setting Match Position - `@_3+ @_3- @_3`

You can "back up" and continue matching from a prior match position using `@` followed by a match variable previously set. E.g.

```
u: ( _~pronoun * go @_0+ often )
```

This matches *I often go* but not *I go* Just as `<` sets the position pointer to the start, `@_0+` makes the pattern think it just matched that wildcard at that location in the sentence going forward.

```
s: ( _is _~she ) # for input: is mother going this sets _0 to is and _1 to mother
s: ( @_1+ going ) # this completes the match of is mother going
```

OK. Setting positional context is really obscure and probably not for you. So why does it exist? It supports shared code for pseudo parsing.

You can match either forwards or backwards. Normally matching is always done forwards. But you can set the direction of matching at the same time as you set a position. Forward matching is `@_n+` and backward matching is `@_n-`.

Note when using backwards matching, `<` and `>` flip meanings. `>` means start at the end (since you are moving backwards) and `<` means confirm we are at start.

`@_3` is a special positional matching called an anchor. It not only makes the position that given and matching forward thereafter, but it also acts as an item that must itself be matched.

E.g., for this pattern `( @_3 is my @_4 life)`

The position pointer moves to `@_3` because as the opening element it can match anywhere in a sentence, just like a word would. But after it matches (example at word 2) then `is` must be word 3 and `my` must be word 4 and `@_4` must start at word 5 and after it completes then `life` must be the next word. Whereas `(< @_3 is)` implies that `@_3` is at position 1, since `<` says this is sentence start.

Backward Wildcards

You can request n words before the current position using `*-n`. For example

```
u: ( I love * > _*-1 ) capture last word of sentence
```

Gory details about strings

'Strings in Output'

A double quoted string in output retains its quotes.

```
u: () I love "rabbits"
```

will print that out literally, including the double quotes.

And you cannot run the string across multiple lines.

An active string interprets variable references inside. It does not show the containing quotes around the whole thing. And it can be extended across multiple lines (treating line breaks as a single space in the string created).

```
u: ( I "take charge" ) OK.
```

When you use "take charge" it can match taking charges, take charge, etc. When you use *"taking charge"* it can only match that, not *"taking charges"*.

If all words in the string are canonical, it can cover all forms. If any are not, it can only literally match.

The quote notation is typically used in several situations...

- you are matching a recognized phrase so quoting it emphasizes that
- what you are matching contains embedded punctuation and you don't want to think

About how to tokenize it e.g., *"Mrs. Watson's"* – is the period part of Mrs, is the `'` part of *Watson*, etc. Easier just to use quotes and let the system handle it.

- You want to use a phrase inside `[]` or `{}` choices. Like `[ like live "really enjoy" ]`

In actuality, when you quote something, the system generates a correctly tokenized set of words and joins them with underscores into a single word. There is no real difference between *"go back"* and *go_back* in a pattern.

But compared to just listing the words in sequence in your pattern, a quoted expression cannot handle optional choices of words. You can't write `go {really almost} back` where that can match *go back* or *go almost back*.

So there is that limitation when using a string inside `[ ]` or `{ }`. But, one can write a pattern for that. While `[ ]` means a choice of words and `{ }` means a choice of optional words, `( )` means these things in sequence. So you could write:

```
u: ( [ next before (go {almost really} back) ] )
```

and that will be a more complex pattern. One almost never has a real use for that capability, but you use `( )` notation all the time, of course. In fact, all rules have an implied `< *` in front of the `()`.

That's what allows them to find a sequence of words starting anywhere in the input. But when you nest `( )` inside, unless you write `< *` yourself, you are committed to remaining in the sequence set up.

As a side note, the quoted expression is faster to match than the `( )` one. That's because the cost of matching is linear in the number of items to test. And a quoted expression (or the `_` equivalent) is a single item, whereas (take charge) is 4 items.

So the first rule will below will match faster than the second rule:

```
u: ( "I love you today when" )

u: ( I love you today when )
```

But quoted expressions only work up to 5 words in the expression (one rarely has a need for more) whereas `()` notation takes any number. And using quotes when it isn't a common phrase is obscure and not worth doing.

Generalizing a pattern word

When you want to generalize a word, a handy thing to do is type :concepts word and see what concepts it belongs to, picking a concept that most broadly expresses your meaning.

The system will show you both concepts and topics that encompass the word. Because topics are more unreliable (contain or may in the future contain words not appropriate to your generalization, topics are always shown a `T~` rather than the mere `~name`.

The deep view of patterns

You normally think of the pattern matcher as matching words from the sentence. It doesn't. It matches marks. A mark is an arbitrary text string that can be associated with a sentence location. The actual words (but not necessarily the actual case you use) are just marks placed on the actual locations in the sentence where the words exist.

The canonical forms of those words are also such marks. As are the concept set names and topic names which have those words as keywords. And all parser determined pos-tag and parser roles of words.

It gets interesting because marks can also cover a sequential range of locations. That's how the system detects phrases as keywords, idioms from the parser like I am a little bit crazy (where a little bit is an adverb covering 3 sentence locations) and contiguous phrasal verbs.

And there are functions you can call to set or erase marks on your own (`^mark` and `^unmark`).

Pattern matching involves looking at tokens while having a current sentence location index. Unless you are using a wildcard, your tokens must occur in contiguous order to match. As they match, the current sentence location is updated. But not necessarily updated to the next adjacent location. It will depend on the length of the mark being matched.

So your token might be `~adverb` but that may match a multiple-word sequence, so the location index will be updated to the end of that index.

And you can play with the location index itself, setting it to the location of a previously matched `_` variable and setting whether matching should proceed

forwards or backwards through the sentence. Really, the actual capabilities of pattern matching are quite outrageous.

Pattern matching operates token by token. If you have a pattern:

```
u: ( {blue} sky is blue )
```

and you input *the sky is blue*, this pattern will fail, even though the initial `{blue}` is optional.

Optional should not lead a pattern, it is used for word alignment. The first blue is found and so the system locks itself at that point. It then looks to see if the next word is sky. It's not. Pattern fails.

It then unlocks itself and allows trying to match from the start of the pattern one later. So `{blue}` matches blue at the end of the sentence. It locks itself. The next word is not sky. Pattern fails. There is nothing left to try.

Interesting thing about match variables

Unlike user variables, which are saved with users, match variables (like `_0`) are global to ChatScript.

They are initialized on startup and never destroyed. They are overwritten by a rule that forces a match value onto them and by things like

```
_0 = ^burst(...)
```

or

```
_3 = ^first(@0all)
```

And those may overrun the needed number of variables by 1 to indicate the end of a sequence. But this means typically `_10` and above are easily available as permanent variables that can hold values across all users etc.

This might be handy in a server context and is definitely handy in `:document` mode where user memory can be very transient. Of course remembering what `_10` means as a holding variable is harder, unlike the ones bound to matches from input. So you can use

```
rename: _bettername _12
```

in a script before any uses of `_bettername`, which now mean `_12`.

Also, although whenever you start to execute a rule's pattern the match variables start memorizing at _0, you can change that. All you have to do is something like this:

```
u: (^eval(_9 = null)  I love _~meat)
```

The act of setting _9 to a value automatically makes the system set the next variable, so future memorizations start at _10. Equivalently, there is `^setwildcardindex`

```
u: (^setwildcardindex(_10 )  I love _~meat)
```

Precautionary note about [ ] and pattern matching retries

When you list a collection of words or concepts within a `[ ]` in a pattern, the system does not try each in turn to find the earliest match. Instead it tries each in turn until it finds a match. Then `[ ]` quits. So if you have a pattern like:

```
u: ( the * [bear raccoon] ate )
```

and an input like *the raccoon at the bear*, then matching would proceed as

- find the word the (position 1 in sentence)
- try to find bear starting at position 2 – found at position 5
- try to find the word ate starting at position 6 – fails

The system is allowed to backtrack and see if the first match can be made later. So it will try to find the later than position one. It would succeed in relocating it to position 4.

It would then try to find the word bear afterwards, and succeed at position 5. It would then try to find the word ate after that and fail. It would retry trying to reset the pattern to find the after position 4 and fail. The match fails.

You can fix this ordering problem by making a concept set of the contents of the `[ ]`, and replacing the `[ ]` with the name of the concept set.

A concept set being matched in a pattern will always find the earliest matching word. The number of elements in a concept set is immaterial both to the order of finding things and to the speed of matching.

ADVANCED OUTPUT

Committed Output

Simple output puts words into the output stream, a magical place that queues up each word you write in a rule output. What I didn't tell you before was that if the rule fails along the way, an incomplete stream is canceled and says nothing to the user. For example,

```
t: I love this rule. ^fail(RULE)
```

Processing the above gambits successively puts the words *I*, *love*, *this*, *rule*, . into the output stream of that rule.

If somewhere along the way that rule fails (in this case by the call at the end), the stream is discarded. If the rule completes and this is a top level rule, the stream is converted into a line of output and stored in the responses list.

When the system is finished processing all rules, it will display the responses list to the user, in the order they were generated (unless you used `^preprint` or `^insertprint` to generate responses in a different order). If the output was destined for storing on a variable or becoming the argument to a function or macro, then the output stream is stored in the appropriate place instead.

I also didn't tell you that the system monitors what it says, and won't repeat itself (even if from a different rule) within the last 20 outputs. So if, when converting the output stream into a response to go in the responses list, the system finds it already had such a response sent to the user in some recently earlier volley, the output is also discarded and the rule "fails".

Actually, it's a bit more complicated than that. Let's imagine a stream is being built up. And then suddenly the rule calls another rule (`^reuse`, `^gambit`, `^respond`). What happens? E.g.

```
u: ( some test ) I like fruit and vegetables. ^reuse(COMMON) And so do you.
```

What happens is this- when the system detects the transfer of control (the `^reuse` call), if there is output pending it is finished off (committed) and packaged for the user. The current stream is cleared, and the rule is erased (if allowed to be). Then the `^reuse()` happens. Even if it fails, this rule has produced output and been erased.

Assuming the reuse doesn't fail, it will have sent whatever it wants into the stream and been packaged up for the user. The rest of the message for this rule now goes into the output stream *and so do you* and then that too is finished off and packaged for the user. The rule is erased because it has output in the stream when it ends (but it was already erased so it doesn't matter).

So, output does two things. It queues up tokens to send to the user, which may be discarded if the rule ultimately fails. And it can call out to various functions. Things those functions may do are permanent, not undone if the rule later fails.

There is a system variable `%response` which will tell you the number of committed responses. Some code (like Harry's control script) do something like this:

```
$_response = %response
...
if ($_response == %response) {...}
```

which is intented to mean if no response has been generated so far, try the code in the ^if. But you have to be wary of the pending buffer. Calling some code, even if it fails, may commit the pending buffer. If there is a chance you will have pending output, the correct and safe way to code this is:

```
^flushoutput()
```

```
$_response = %response
...
if ($_response == %response) {...}
```

`^flushoutput` will commit pending output.

Output cannot have rules in it

Output script cannot emed another rule inside it. Output is executed during the current volley whereas rules (like rejoinder rules) may be executed in a different volley. Therefore this is illegal:

```
u: GREETING ( ~emohello )
   if ($username)
   {
     Hi  $username!
   }
   else
   {
     I don't believe we've met, what's your name?
     a: (_*) So your name is '_0?
   }
```

and needs to be written like this:

```
u: GREETING ( ~emohello )
   if ($username)
   {
     Hi  $username!
   }
   else
   {
     I don't believe we've met, what's your name?
   }
  a: (_*) So your name is '_0?
```

Of course you don't want the rejoinder triggered if you can from the `if` side, so you'd also need to add a call to `^setnorejoinder` from inside it.

Formatted double quotes (Active/Format String)

Because you can intermix program script with ordinary output words, ChatScript normally autoformats output code. But programming languages allow you to control your output with format strings and ChatScript is no exception.

In the case of ChatScript, the active string `^"xxx"` string is a format string. The system will remove the `^` and the quotes and put it out exactly as you have it,

except, it will substitute variables (which you learn about shortly) with their values and it will accept `[ ] [ ]` choice blocks. And will allow function calls. You can't do assignment statements or loops or if statements.

```
t: ^"I like you."
```

puts out I like you.

When you want special characters in the format string like ", you need to backslash them, which the format string removes when it executes. For example:

```
u: () $tmp = [ hi ^"there \"john\"" ]
```

the quote inside the format string need protecting using `\"`.

You can write `\n`, `\r`, `\t` and those will be translated into newline, carriage return, and tab. However, you should avoid `\r` because on LINUX it is not needed and in Windows the system will change `\n` to carriage-return and newline.

Format strings evaluate themselves as soon as they can. If you write:

```
u: () $tmp = ^" This is $var output"
```

then `$tmp` will be set to the result of evaluating the format string. Similarly, if you write:

```
u: () $tmp = ^myfunc(^" This is $var output")
```

then the format string is evaluated before being sent to `^myfunc`.

You can continue a format string across multiple source lines. It will always have a single space representing the line change, regardless of how many spaces were before or after the line break. E.g

```
^"this is my life" # -> ^"this is my life"
```

regardless of whether there were no spaces after is or 100 spaces after is. You may not have comments at the ends of such lines (they would be absorbed into the string).

While an active string tries to detect and substitute variables, it can't succeed if you have letters immediately after the variable name. E.g.

```
^"T$$time_computed_hrs:$$time_computed_mins:$$time_computed_secsZ"
```

The Z at the end of the `$$time_computed_secs` will make that variable hard to detect, since it will look like the variable is `$$time_computed_secsZ`.

You can fix that by escaping the next character not part of the name. `$$time_computed_secs\Z`.

Json Active Strings

`^' xxxxxx '` is another kind of active string. It is intended for writing easy JSON, because you don't have to escape doublequotes unless json would need to.

It will converting `\n`,`\t`, `\r` into their control characters, convert `\\` into just `\`, and leave all other characters alone. Eg

```
^'{ "test" : "my \"value\" \x" }
```

will become

```
^'{ "test" : "my \"value\" \x" }
```

Functional Strings

Whenever format strings are placed in tables, they become a slightly different flavor, called functional strings. They are like regular output- they are literally output script.

Formatting is automatic and you get to make them do any kind of executable thing, as though you were staring at actual output script. So you lose the ability to control spacing, but gain full other output execution abilities.

They have to be different because there is no user context when compiling a table. As a consequence, if you have table code that looks like this:

```
^createfact( x y ^" This is $var output")
```

the functional string does NOT evaluate itself before going to createfact. It gets stored as its original self.

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

Loop Construct – loop or ^loop

Loop allows you to repeat script. It takes an optional argument within parens, which is how many times to loop. It executes the code within `{ }` until the loop count expires or until a `FAIL` or `END` code of some kind is issued.

- `End(loop)` signals merely the end of the loop, not really the rule, and will not cancel any pending output in the output stream.
- `Fail(LOOP)` will terminate both loop and rule enclosing. All other return codes have their usual effect.

Deprecated is `fail(rule)` and `end(rule)` which merely terminated the loop.

```
t: Hello. ^loop (5) { me }

t: ^loop () { This is forever, not. end(LOOP)}
```

The first gambit prints *Hello. me me me me me.* The second loop would print forever, but actually prints out This is forever, not. because after starting output, the loop is terminated.

Loop also has a built in limit of 1000 so it will never run forever. You can override this if you define `$cs_looplimit` to have some value you prefer.

`^loop( n )`

Loop can be given a count. This can be either a number, function call that results in a number, or you can use a factset id to loop through each item of the factset via

^loop (@0)

If Construct - `if` or `^if`

The if allows you to conditionally execute blocks of script. The full syntax is:

```
if ( test1 )
    {
      script1
    }
else if ( test2 )
    {
      script2
    }
# ...
else
    {
      script3
    }
```

You can omit the else if section, having just if and else, and you can omit the else section, having just if or if and else if. You may have any number of else if sections. The test condition can be:

- A variable – if it is defined, the test passes
- `!` variable – if it is not defined, the test passes (same as relation variable `==` null)
- A function call – if it doesn't fail and doesn't return the values 0 or false, it passes
- A relation – one of `== != < <= > >= ? !?`

For the purposes of numeric comparison (< <= > >=) a null value compared against a number will be considers as 0.

You may have a series of test conditions separated by AND and OR. The failure of the test condition can be any end or fail code. It does not affect outside the condition; it merely controls which branch of the if gets taken.

```
if ($var) { } # if $var has a value

if ($var == 5 and foo(3)) {} # if $var is 5 and foo(3) doesn't fail or return 0 or false
```

Comparison tests between two text strings is case insensitive.

A word of warning on the ? (in set) relation test. It only works for actual concepts that have enumerated values. A number of sets marked by the engine for patterns do not consist of enumerated members.

All of the pos-tagging and parse-related concepts are like this, so you cannot use ~number, ~noun, ~verb, etc here.

It will work if you compare a match variable derived from input, because that has access to knowing all the marked concepts of that particular word.

Pattern If

An alternative If test condition is the pattern If. You write the test using the word pattern at the start, and then you write exactly what you can write when you write a rule pattern. Eg.

```
if (pattern bingo _*1 ~helo) { ... }
```

This gives you the full power of the pattern matcher, including the ability to match and memorize from the current input.

Quoting

Normally output evaluates things it sees. This includes $user variables, which print out their value. But if you put quote in front of it, it prints its own name. '$name will print *$name*.

The exception to this rule is that internal functions that process their own arguments uniquely can do what they want, and the query function defines '$name to mean use the contents of $name, just don't expand it if it is a concept or topic name as value.

Similarly, a function variable like ^name will pretend it was its content originally. This means if the value was $var then, had $var been there in the output originally, it would have printed out its content. So normally ^name will print

out the contents of its content. Again, you can suppress with using `'^name` to force it to only print its content directly.

Outputting underscores

Normal English sentences do not contain underscores. Wordnet uses underscores in composite words involving spaces.

ChatScript, therefore has a special use for underscores internally and if you put underscores in your output text, when they are shipped to the user they are converted to spaces.

This doesn't apply to internal uses like storing on variables. So normally you cannot output an underscore to a user. But a web address might legitimately contain underscores. So, if you put two underscores in a row, ChatScript will output a single underscore to the user.

Response Controlinput

Having said that CS automatically changes underscores to spaces, you can alter this and other default response output processing. The variable `$cs_response` can be set to some combination of values to alter behavior. The default value is

```
$cs_response = #RESPONSE_UPPERSTART +
               #RESPONSE_REMOVESPACEBEFORECOMMA +
               #RESPONSE_ALTERUNDERSCORES
```

which controls automatically up-casing the first letter of output, removing spaces before commas, and converting underscores to spaces (and also removing ~ from concept names).

Equivalently

```
$cs_response = #ALL_RESPONSES
```

which if you want all is what you should use in case new ones are added in the system later.

Output Macros

Just as you can write your own common routines (functions) for handling pattern code with patternmacro:, you can do the same for output code.

```
Outputmacro: name (^arg1 ^arg2 ...)
```

and then your code. Only now you use output script instead of pattern stuff. Again, when calling the macro, arguments are separated with spaces and not commas.

Whereas most programming language separate their arguments with commas because they are reserved tokens in their language, in ChatScript a comma is a normal word. So you separate arguments to functions just with spaces.

```
?: ( hi) ^FiveArgFunction( 1 3 my , word)
```

Outputmacros can return a value, just like a normal function. You just dump the text as you would a message to the user.

```
outputmacro: ^mymac()
tested here

TOPIC: ~patterns keep repeat []
#! what time is it?
u: ( << what time >> ) $test = ^mymac() join(ok $test)
```

will print *oktested here*.

However, it is clearer and cleaner if you are returning data to be stored somewhere else (not to be merely immediately sent to the user), to use `^return("tested here")`. This both creates the result, and ends the function immediately even if other code follows.

Note - calls to macros use "pass by reference", so the actual value of the `^variable` is the name of what was passed in, and it is generally (but not always) evaluated on use.

You may make references to outputmacros before they are defined, EXCEPT when the Function is directly or indirectly referenced from a table. Tables immediately execute as they are compiled, and you will get an error if a function it tries to use is not defined.

Indirect function calls

You can store an outputmacro name on a variable and then call that indirectly.

```
^$_xx(value1)  if $_xx holds a function name
```

Sharing function definitions

ChatScript requires that a function be defined before use. When you use that function from multiple files, you may have trouble ordering the files for compilation if you merely name the folder in `filesxxx.txt` since you cannot guarantee compilation order unless you explicitly name the files. But you can also just put your functions in a top level file and then have your other files in folders, and name it and then them in your `filesxxx.txt` file.

Save-Restore locals

`$$xxx` and `$xxx` variables are global, merely transient and permanent.

Function variables like `^myval` are restricted in use to the function declaring them, so they are sort of local variables, but they are stand-ins for the arguments passed, which means if you write indirectly the function variable you are changing something above you as well.

Fortunately there are local variables, `$_xxx`.

Without local variables, it is easy to accidentally reuse the same name of a transient variable that you used above you in the call chain. Imagine this:

```
outputmacro: ^mycall()
    $$counter = 0
    loop()
        {
        # ...
        $$counter += 1
        }
```

and this:

```
$$counter = 0
loop()
    {
    $$tmp = ^mycall()
    # ...
    $$counter += 1
    }
```

You have two areas using the same counter variable and the inner one destroys the outer one. Here is where save-restore variables come in.

You can either use local variables, when you don't need to pass information between places except via function args. Or you can use save-restore variables.

You can declare a list of variables whose contents will be memorized on entry to a function or a topic, and restored to their former values on exit.
You can safely write all over them within the function or topic, without harming a caller. And they are still global, in that they are visible to anyone your function calls.

Of course if you intend to pass back data in a global variable, don't put it in your save-restore list.

```
Outputmacro: ^myfunc(^arg1)($$tmp  $global  $$tmp2) # $$bestscore exported
# code

Topic: ~mytopic(keyword1 keyword2) ($$tmp $$global $$tmp2)
```

```
# rules
```

You can protect both transient and permanent variables, but usually you would just protect all of the transient variables you assign values to inside your function or topic. The comment is what I would say if I intended a variable be returned outside in addition to a primary return value. That way anyone reading the code would know $$bestscore was not accidentally left off the save-restore list.

And whenever you can, prefer local variables because then you don't have to remember to add them to the protected save-restore list. ChatScript does that automatically.

```
Outputmacro: ^myfunc(^arg1)($$tmp $global $$tmp2) # $$bestscore exported ... code
```

You can protect both transient and permanent variables, but usually you would just protect all of the transient variables you assign values to inside your function.

The comment is what I would say if I intended a local variable be returned outside in addition to a primary return value. That way anyone reading the code would know `$$bestscore` was not accidentally left off the save-restore list.

```
dualmacro: name(...)
```

is exactly like `outputmacro:`, but the function can be called on then side or on the output side. Typically this makes most sense for a function that performs a fixed `^query` which you can see if it fails in pattern side or as a test on the output side or inside an if condition.

Output macros can be passed system function names and output macro names, allowing you to indirectly call things. E.g.

```
outputmacro: ^indirect(^fn ^value)
    $$tmp = ^fn(^value)
```

The above will evaluate `^fn`, and if it finds that it is a function name, will use that in a call. The only tricky part is creating the name of the function to call in the first place.

If you just write a function name with ^ in output, the compiler will complain about your call. So you have to synthesize the name somehow. Here are two ways:

```
outputmacro: ^mycall()
    $$tmpfn = ^join( ^"\^" func)
    ^indirect($$tmpfn 34)
    ^indirect( ^"\^func" 34)
```

You can also store function names on user and match variables and then call them. E.g.

```
$$tmp = ^"\^func"
$$tmp(34)
```

You can declare an outputmacro to accept a variable number of arguments. You define the macro with the maximum and then put "variable" before the argument list. All missing arguments will be set to null on the call.

```
outputmacro: ^myfn variable (^arg1 ^arg2 ^arg3 ^arg4)
```

Output Macros vs `^reuse()`

An outputmacro is a block of code, treated as a kind of function. But another way to make a block of code is create a rule and `^reuse` it. E.g.

```
s: MYCODE ( ? ) here is a block of code that goes on and on
```

Code you write in an output macro could be code you write on the output side of a `^reused` rule. Notice that this rule can never trigger on its own (an input sentence cannot be a statement and a question simultaneously).

So what are the distinctions between the two? The distinction is not in tracing. You can trace a single rule or an outputmacro equally. And both return to their caller when normally complete.

An advantage for outputmacros is that you can pass them arguments, making them more easily tailorable. To do the same with a rule, you have to store values on globals, which is more inconvenient, harder to understand, and subject to the risk that you accidentally reuse the same variable in something the rule calls. Similarly outputmacros can return a value, while you have to use globals to return a value from a rule.

An advantage for rules is that they can have rejoinders. So if the rule generates output, it may also have rejoinders to react to it. Another advantage for rules is that you can terminate their execution early `^end(RULE)` without impacting the calling rule. There is no such ability in an outputmacro, so you'd have to organize if statements to manage early termination effects.

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.

Randomized Output Revisited `[ ]`

Remember this construct:

```
?: ( hi ) [How are you feeling?][Why are you here?]
```

These choices are picked equally. But maybe you don't want some choices. You can put an existence test of a variable at the start of a choice to restrict it.

```
?: ( hi ) [$ready How are you feeling?][Why are you here?]
```

In the above, the first choice is controlled by `$ready`. If it is undefined, the choice cannot be used. You can also use negative tests.

```
?: ( hi ) [!$ready this is a][This is b]
```

In the above only if `$ready` is undefined can you say *this is a*

If you want the variable to be a lead item in the actual output of a choice, you can do this:

```
?: (hi) [^eval($ready) is part of the output]
```

or the more clunky:

```
?: ( hi ) _0 = $ready [ _0 is part of the output]
```

Choices lead to issues when you want rejoinders. You can label rejoinder branches of choices. Those without labels default to `a:`

```
?: ( what meat ) [c: rabbit ] [e: steak] [ h: lamb] [pork]
    a: this rejoinders pork
    c: this rejoinders rabbit
    e: this rejoinders steak
    f: this rejoinders on e:
    h: this rejoinders lamb
```

In the above, pork rejoinders at `a:`, while the other choices name their rejoinder value. Each new starting label needs to be at least one higher than the rejoinder before it. That allows the system to detect rejoinders on rejoinders from choice branches.

If you do both variable control and rejoinder label, the control comes first and label after you have successful control.

```
?: ( what meat ) [$ready c: rabbit ] [e: steak] [ g: lamb] [pork]
```

Advanced Variables

Local Variables

User variables `$xxx` and `$$xxx` are all global in scope.

Anyone can access them to get or set them.

But if you want to write safe code, you also want local variables that no one can access and ruin on you. `$_xxx` are local variables.

When you use them inside an outputmacro or a topic, they only have meaning and access inside that code. No other code can see them. They can use the same variable name, and get their own local instance.

Local variables always start out initialized to null. If you pass them to an outputmacro, unlike normal variables which are passed by reference (hence can be written upon if you use Indirection Variables below), local variables are passed by value. Their content is passed, so no one can touch the variable itself.

Local variables used in a topic all remain accessible while in that topic. If you do `^gambit(~topic)` and then inside you do `^gambit(~)` or `^reuse(MYLABEL)`, you have not left the topic and so the variables remain intact. But if you call outside the topic, then a deeper call to the original topic sees new variables (just as a recursive call to an outputmacro would).

Hence `^gambit(~mytopic)` which calls `^reuse(~histopic.label)` which then calls `^gambit(~mytopic)` will get two different instances of `~mytopic` and the variables separately handled by each.

By definition, local variables are transient and do not get saved in a user's topic file.

Indirection Variables

You can, of course, merely refer to a variable in a script to set or get its value. But suppose you wanted to associate a variable with every topic you have.

Maybe when the user says *I'm bored* you want to leave a topic and not reopen it yourself for any near future (say 200 volleys).
In that case, you don't have a variable but you need to create one dynamically.

```
$$topicname = substitute(character %topic ~ "")
```

The above gives you a topic name without the `~`. Pretend the current topic is `~washing`. Then you can create a dynamic variable name simply by using

```
$$tmp = ^join($ $$topicname)
```

which would create the name $washing.

Now suppose you wanted to set that variable to some number representing the turn after which you might return.

```
$$tmp = 25
```

doesn't work. It wipes out the *washingvalueof*'$tmp`and replaces it with`25'.

You can set indirectly through `$$tmp` using function notation `^$$tmp` The above says take the value of `$$tmp`, treat it as the name of a variable, and assign into it. Which means it does the equivalent of

```
$washing = 25
```

If you want to an indirect value back, you can't do:

```
$$val = $$tmp
```

because that just passes the name of `$washing` over to `$$val`. Instead you do indirection again:

```
$$val = ^$$tmp # $$val becomes 25
```

Indirection works with values that are user variables and with values that are match variables or quoted match variables.

Many routines automatically evaluate a variable when using it as an argument, like `^gambit($$tmp)`. But if you want the value of the variable it represents, then you need to evaluate it another time.

```
$$tmp1 = ^$$tmp
^gambit($$tmp1)
```

Equivalently `^gambit(^$$tmp1)` is legal.

See also Indirect Pattern Elements in ChatScript Pattern Redux manual.

Bot variables

You can also define variables that are "owned" by the bot. Any variables you create as part of layer 0 or layer 1 are always resident. As are any variables you define on the command line starting ChatScript. You can see them and even change them during a volley, but they will always refresh back to their original values at the start of the next volley.

Match Variables

Match variables like `_5` are generally the result of using an underscore in a pattern match. Match variables hold 3 pieces of data

- original word(s)
- canonical word(s)
- position, range location of the word(s).

You can transfer part of Assigning match variables to user variables:

```
$$stuff = _0
```

but user variables only have a single piece of data. So on assignment you lose 2 of the 3 pieces from the match variable. You can choose which words (original or canonical) when you assign.

```
$$stuff = '_0 #  original words
```

You can store positional data onto a different variable using `^position(start _0)` or

```
^position(end _0).
_0 = $$stuff
```

When you assign onto a match variable from a user variable, you make both original and canonical values of the match variable the same, and the positional data is set to 0.

```
_0 = _10
```

This is a transfer from one match variable to another, so no data is lost.

One unusual property of match variables is that they are not cleared between volleys. This makes them the ONLY way you can pass data between volleys on a server where different users are involved.

Note: Match variables have a 20,000 character limit.

JSON dotted notation for variables

If a variable holds a JSON object value, you can directly set and get from fields of that object using dotted notation.

This can be a fixed static fieldname or a variable value- `$myvar.$myfield` is legal.

Dotted notation is cleaner and faster than `^jsonpath` and `jsonobjectinsert` and for get, has the advantage that it never fails, it only returns null if it can't find the field. On the other hand, assignment fails if the path does not contain a json object at some level.

```
$x = $$$obj.name.value.data.side
$$$obj.name.value.data.side = 7
```

Assigning `null` will remove a JSON key entirely. Assigning `"" ^""` will set the field to the JSON literal `null`.

Fact dotted notation for variables

If `$$f` holds a fact id, then

```
$$f.subject
$$f.verb
$$f.object
```

will return those components. You may NOT, however, assign into them.

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 insure 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.
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. 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.

`^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`

Build warning messages

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

```
:build 0 nospell
```

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

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

```
:build 0 outputspell
```

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

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

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

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

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

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

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

Since you have accepted all forms of cheese, you don't need to name `cheese~n`. `: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 character from scratch (equivalent to `:reset user`).

Build Layers

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

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

- `:build common0` – shared data-source (level 0)
- `:build george` – george bot-specific (level 1)
- `:build henry` – henry bot-specific (level 1)
- `:build all` – does george and henry and others (level 1)
- `: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 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 <<## :

```
<<##SPECIAL ...
... >>##
```

```
: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.

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

file	description
`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.
`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 script that resides 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 insure the topic has appropriate keywords if needed.

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

What really happens with rule erasure?

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

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.

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]