ChatScript System Functions Manual

Revision 1/26/2020 cs10.0

System functions are predefined and can be intermixed with direct output. Generally they are used from the output side of a rule, but in many cases nothing prevents you from invoking them from inside a pattern. When used in a pattern, they do not write out any text output to the user. But their output will be tested the same as it would from an `if` statement, meaning 0 and false are failures.

You can write them with or without a ^ in front of their name in output. With is clearer, but you don't have to. The only times you must is if the first thing you want to do in a gambit is call a function (unlikely) or you are in a pattern.

```
t: name(xxx)
u: ( [find (me go) ])
```

These are ambiguous. In the gambit is it function call or label. In the responder is find a function name or just a word?

The above gambit is treated as a label and pattern. You can force it to be a function call by one of these:

```
t: ^name(xxx)   # explicilty say it is a function

t: () name(xxx) # explicitly add an empty pattern
```

Rule Tags

Some functions out or take "rule tags". All rules have an internal label consisting of `~topic.toplevelindex.rejoinderindex`. E.g.

```
~introductions.0.5
```

stands for the 0th rule in the `~introductions` topic, rejoinder #5.

Topic Functions

^addtopic (topicname)

adds the named topic as a pending topic at the head of the list. Typically you don't need to do this, because finding a reaction from a topic which is not a system, disabled, or nostay topic will automatically add the topic to the pending list. Never returns a fail code even if the topic name is bad.

^available (ruletag optionalfail)

Sees if the named rule is available (1) or used up (0). If you supply the optional argument, the function will fail if the rule is not available.

^cleartopics()

Empty the pending topics list.

^counttopic (topic what)

For the given topic, return how many rules match what.

What is `gambits`, `responders`, `availablegambits`, `availableresponders`,`rules`, `used`.

That is, how many gambits exist, how many available gambits exist (not erased), how many responders exist, how many available responders exist (not erased), how many top level rules (gambits + responders) exist, and how many top level rules have been erased.

^gambit (value value ...)

If value is a topic name, runs the topic in gambit mode to see if any gambits arise. If none arise from the first value, it will try the second, and so on. It does not fail unless a rule forces it to fail or the named topic doesn't exist or is disabled. You can supply an optional last argument `FAIL`, in which case it will return `FAILRULE_BIT` if it didn't fail but it didn't generate any new output either.

The value may be `~`, which means use the current topic you are within. It can also be `PENDING`, which means pick a topic from the pending topics stack (they are all pending being returned to but not including the current topic). Or it can be any other word, which will be a keyword of some topic to pick. E.g.,

```
^gambit(~ PENDING ~mygeneraltopic FAIL)
```

^findrule (label)

On the assumption that you only have one occurence of a rule label in your script, if you provide that label to this function, it will find the corresponding rule anywhere in your script and return the rule tag corresponding to it. If you have more than one such labelled Rule it merely returns the first one it finds (which will be earliest rule in earliest compiled topic).

^getrule (what label)

for the given rule label or tag, return some fragment of the rule.

what can be `tag`, `type`,`label`, `pattern`, `output`,`topic`, and `usable`.

The type will be `t`, `?`, `s`, `a`, etc.

If a rule label is involved, optional third argument if given means only find enabled rules with that label. For usable, returns 1 if is can be used or null if it has been erased. The label `~` means the current rule. The label `0` means the top level rule above us (if we are a rejoinder, otherwise it is the same as `~`).

^hasgambit (topic)

fails if topic does not have any gambits left unexecuted.

Even it if does, they may not execute if they have patterns and they don't match. Optional second argument, if `any` will return normally if topic has any gambits (executed or not) and will failrule if topic has no gambits (a reactor topic).

^keep()

do not erase this top level rule when it executes its output part (you could declare a topic to be this, although it wouldn't affect gambits).

Doing `keep()` on a gambit is quite risky since gambits after it may not ever fire.

^lastused (topic what)

given a topic name, get the volley of the last what, where *what* is `GAMBIT`, `RESPONDER`, `REJOINDER`, `ANY`. If it has never happened, the value is `0`.

^next (what {label})

Given what of GAMBIT or RESPONDER or REJOINDER or RULE and a rule label or tag, find the next rule of that what. Fails if none is found.

REJOINDER will fail if it reaches the next top level rule.

If *label* is ~, it will use the last call's answer as the starting point, enabling you to walk rules in succession.

There is also ^next(FACT @xxx) - see fact manual.

For ^next(INPUT) the system will read the next sentence and prep the system with it. This means that all patterns and code executing thereafter will be in the context of the next input sentence. That sentence is now used up, and will not be seen next when the current revised sentence finishes.

Sample code might be:

```
t: Do you have any pets
    a: ( ~yes ) refine()
        b: ( %more ) ^next(input) refine()
            c: ( ~pets ) ... # react to pet
            c: () ^retry(SENTENCE) # return to try input from scratch
        b: () What kind do you have?
            c: ( ~pets ) ... # react to pet
```

If *label* is LOOP, the system will stop processing code in the current loop and return to the next iteration of it, e.g. C++/Java continue, except that it will stop all code and return to however high up the loop really is, exiting topics and functions willy nilly if need be.

^poptopic (topicname)

Removes the named topic as a pending topic. The intent is not to automatically return here in future conversation. If topicname is omitted, removes the current topic AND makes the current topic fail execution at this point.

^refine (?)

This is like a switch statement in *C language*. It executes in order the rejoinders attached to its rule in sequence.

When the pattern of one matches, it executes that output and is done, regardless of whether or not the output fails or generates nothing. It does not "fail", unless you add an optional FAIL argument. You can also provide a rule tag. Normally it uses the rule the refine is executing from, but you can direct it to refine from any rule.

`^rejoinder ( {tag/label} )`

Without argument, see if the prior input ended with a potential rejoinder rule, and if so test it on the current sentence. If we match and dont fail on a rejoinder, the rejoinder is satisfied. If we fail to match on the 1st input sentence, the rejoinder remains in place for a second sentence. If that doesn't match, it is canceled. It is also canceled if output matching the first sentence sets a rejoinder.

You can give an optional tag or label to pretend the named rule had been the one to set a rejoinder and so therefore execute its rejoinders explicitly.

`^respond ( value value ... )`

Tests the sentence against the named value topic in responder mode to see if any rule matches (executes the rule when matched). It does not fail (though it may not generate any output), unless a rule forces it to fail or the topic requested does not exist or is disabled.

This rule will not erase but the responding rule might. If the first value fails to generate an answer, it tries the second, and so on. You can supply an optional last argument `FAIL`, in which case it will return `FAILRULE_BIT` if it didn't fail but it didn't generate any new output either. You could instead supply an optional last argument `TEST`, in which case a topic is executed to see if a rule will match. If so, the tag is returned and no output is made from the topic (and no rule is used up).

If a value designates a labelled or tagged rule (e.g., `~mytopic.mylabel` or `~mytopic.1.0`) then the system will skip over all rules until it reaches that rule, then begin linear scanning, even if the topic is designated random.

The *value* may be `~`, which means use the current topic you are within.

It can also be `PENDING`, which means pick a topic from the pending topics stack (they are all pending being returned to but not including the current topic). Or it can be any other word, which will be a keyword of some topic to pick.

`^retry ( item )`

If *item* is `RULE` reexecute the current rule. It will automatically try to match one word later than its first match previously.

If *item* is `TOPIC` it will try the topic over again.

If *item* is `SENTENCE` it will retry doing the sentence again. To prevent infinite loops, it will not perform more than 5 retries during a volley. `SENTENCE` is particularly useful with changing the tokenflags to get input processing done differently. If item is INPUT it will retry all input again.

`^retry(TOPRULE)` will return back to the top level rule (not of the topic but of a rejoinder set) and retry.
It's the same if the current rule was a top level rule, but if the current rule is from `^refine()`, then it returns to the outermost rule to restart. If the current rule is not from `^refine()`, then `TOPRULE` means the lexically placed toprule above the current rule and a `^reuse()` will be performed to go to it.

`^reuse ( rule label optional-enable optional-FAIL )`

Uses the output script of another rule. The label can either be a simple rule label within the current topic, or it can be a dotted pair of a topic name and a label within that topic or it can be a rule tag.

`^reuse` stops at the first correctly labeled rule it can find and issues a RULE fail if it cannot find one. Assuming nothing fails, it will return 0 regardless of whether or not any output was generated.

When it executes the output of the other rule, that rule is credited with matching and is disabled if it is allowed. If not allowed, the calling rule will be disabled if it can be.

```
t: NAME () My name is Bob.

?: ( << what you name >> )
    ^reuse(NAME)

?: ( << what you girlfriend name >> )
    ^reuse(~SARAH.NAME)
```

Normally reuse will use the output of a rule whether or not the rule has been disabled. But, if you supply a 2nd argument (whatever it is), then it will ignore disabled ones and try to find one with the same label that is not disabled. You can also supply a `FAIL` argument (as either 2nd or 3rd) which indicates the system should issue a RULE FAIL if it doesn't generate any output.

If you want to use a common rule to hold an answer and ONLY fire when reused, perhaps with rejoinders, the most efficient way to do that is with a rule whose pattern can never match. E.g. like this:

```
s: COMMON (?) some answer
    a: () some rejoinder...
```

You make `^reuses` go to COMMON (or whatever you name it) or even `^setrejoinder` on it. The rule itself can never trigger because it only considers its pattern when the input is a statement, but the pattern says the input must be a question. So this rule never matches on its own.

There are also a variety of functions that return facts about a topic, but you have to read the facts manual to learn about them.

`^sequence ( ? )`

This is like `^refine`, except instead of only executing the first rejoinder that matches, it executes all matching rejoinders in order. If one of the rule outputs fails, it stops by failing the calling rule.

Normally `^sequence` uses the rejoinders of the rule that it is executing from, but you can direct it to `^sequence` the rejoinders of any rule.

`^setrejoinder ( {kind} tag )`

Force the output rejoinder to be set to the given tag or rule label. It's as though that rule had just executed, so the rules beneath it will be the rejoinders to try.

If *kind* is `input` then the input rejoinder is set.

If *kind* is `output` or is omitted, then it sets the output rejoinder.

`^setrejoinder` does not jump anywhere. It establishes the context for `^rejoinder`.

When you do:

```
t: what is your name
    a: ATX(_~propernoun) Hi, '_0
```

the outputrejoinder is set to `ATX`. You can change that if you want. When the next volley comes in, the outputrejoinder is now the inputrejoinder and used for `^rejoinder`. You can modify that as well. Both can exist simultaneously, you have the input context and you set an output context before having used up the inputrejoinder.

Setting a rejoinder on a rule means starting with the rejoinder immediately after it. If you were trying to copy a rejoinder that had already been established and redo it later, eg.

`^setrejoinder(output %inputrejoinder)`

this would be problematic, because it would set it to the rule after, which would be wrong. For this use the kind of "copy" which does not have issues with this.

`^setrejoinder(copy %inputrejoinder)`

If *kind* is output or copy and the *tag* is `null`, the output rejoinder is cleared (analogous to ^disable).

If the *kind* is input and the *tag* is `null`, the input rejoinder is cleared.

To kill a set outputrejoinder, use ^disable(OUTPUTREJOINDER).

`^topicflags ( topic )`

Given a topic name, return the control bits for that topic. The bits are mapped in dictionary_system.h as `TOPIC_*`.

`^sleep ( milliseconds )`

This stalls the engine for that many milliseconds. If this is a server, the server is unavailable until sleep is done. Use with care. A good use is when starting up a server instance and the boot process involves reading from an API. If your machine runs 30 instances of ChatScript launched at once (to use max CPU), then all of them hitting the same API at once may be bad for the API and forcing a randomized sleep based on processid is a good use.

Marking Functions

`^mark ( word location {ONE ALL})`

Marking and unmarking words and concepts is fundamental to the pattern matching mechanism, so the system provides both an automatic marking mechanism and manual override abilities. You can manually mark or unmark something. Automatic system marking marks all concepts implied by chasing up membership in other concepts, as does this call `^mark`. `word` can be any word, which also means you can mark something with a concept name whether or not the concept actually is defined anywhere.

There are two mechanisms supported using `^mark` and `^unmark`: specific and generic.

With `specific`, you name words or concepts to mark or unmark, either at a particular point in the sentence or throughout the sentence. By default, or using the optional third argument `ALL`, not only is what you name marked, but anything it in turn is a part of is marked. The optional third argument `ONE` will only mark that named word/concept and none of the hierarchy implied by its membership in yet some other concept.

With `generic` you disable or reenable all existing marks on a word or words in the sentence. In fact, you go beyond that because during patttern matching words you disable are invisble entirely, and matching proceeds as if they do not exist.

`Specific`: effects are permanent for the volley and cross over to other rules. In documentation below, use of `_0` symbolizes use of any match variable.

^mark and ^unmark can mark/unmark a single thing, or the hierarchy of things. Mark takes a possible 3rd argument ONE or ALL (defaulted) which makes it mark either just the thing you gave or traverse the hierarchy and mark all of it.

```
^mark ( ~meat _0 )
```

This marks `~meat` as though it has been seen at whereever sentence location `_0` is bound to (start and end)

```
^mark ( ~meat n )
```

Assuming `n` is within 1 and sentence word limit, this marks meat at nth word location. If `n` was gotten from ^position of a match variable, it is the range of that match variable.

```
^mark ( tomboy _0 )
```

This marks the word tomboy as visible at the location designated, even though this word is not actually in the sentence. While patterns will react to its presence, it will not show up in any memorizations using `_`.

While usually you mark a concept, you can also mark a word (though you should generally use the canonical form of the word to trigger all its normal concept hierarchy markings as well).

Although `^conceptlist` (see Facts manual) normally only reports concepts marked at a word, if you explicitly mark using a word and not a concept, that will also be reported in `^conceptlist`.

```
^mark ( ~meat )
```

With location omitted, this marks `~meat` as though it has been seen at sentence start (location 1).

```
^mark()
```

Clears all global unmarks. restore a global ^unmark(0) exactly as it was before the global unmark.

^unmark (word _0)

The inverse of specific **^mark**, this takes a matchvariable that was filled at the position in the sentence you want erased and removes the mark on the word or concept set or topic name given. Pattern matching for it in that position will now fail. If the word was a phrase, then all words in that phrase have the mark removed. Thus **South Georgia** which has **Georgia** embedded within it, and both might be ~geographic_area, will have both words unmarked if you unmark ~geographic_area. But it is not symmetric to **^mark** because it does not remove all implied marks that mark may have set.

If you end up calling ^unmark with a 2nd argument of null, it will just return without failure. This can happen if you pass null to an outputmacro: myfn(var) and then ^unmark(xxx ^var).

^Unmark can remove either the single thing, or ^unmark(* _0) removes ALL marks and makes the word completely invisible (not seen in a wildcard match), whereas ^unmark(@ _0) removes all marks but leaves the word occupying space in the sentence and will be seen in a wildcard match.

When you call unmark, the analysis of the sentence HAS ALREADY HAPPENED.
If a ~noun was detected, so has ~noun_and_some_concept. So if you then erase ~noun mark, it has NO impact on other marks made.

^unmark (* n)

Assuming **n** is within **1** and sentence word limit, this unmarks all concepts at nth word location. If n was gotten from **^position** of a match variable, it is the range of that match variable.

^unmark (word all)

All references to word (or **~concept** if you named one) are removed from anywhere in the sentence.

Generic: effects are transient if done inside a pattern, last the volley if done in output. When you are trying to analyze pieces of a sentence, you may want to have a pattern that finds a kind of word, notes information, then hides that kind of word and reanalyzes the input again looking for another of that ilk.

Being able to temporarily hide marks can be quite useful, and this means typically you use **^unmark** of some flavor to hide words, and then **^mark** later to reenable access to those hidden words.

`^unmark ( * _0 )`

Turns off ALL matches on this location temporarily. The word becomes invisible and takes up no space in the sentence. It disables matching at any of the words spanned by the match variable. This unmark will also block subsequent specific marking using `^mark` at their locations.

`^unmark ( @ _0 )`

Turns off ALL matches on this location range but keep the word visible. It cannot be matched by anything but a wildcard and takes up its space in the sentence. Often this is used before something like 'mark(~city _0) where you are taking something with ambiguous meanings, like "nice", and removing all wrong meanings (and right ones) and putting back just right ones. This is a way for the rest of processing to only see the correct interpretation of a word.

Turns off matching on all words of the sentence.

`^mark ( * _0 )`

To restore all marks to some location after having used ^unmark(* _0)

`^unmark ( * )`

Turn off all words of the sentence. Probably not that useful.

`^mark ( * )`

Restores all marks of the sentence, for words that had ^unmark(* _0) performed.

Reminder: If you do a generic unmark from within a pattern, it is transient and will be turned off when the pattern match finishes (so you don't ruin later rules), whereas when you do it from output, then the change persists for the rest of the volley. Furthermore it is handy to flip specific collections of generic unmarks on an off.

`^mark()` memorizes the set of all * unmarks (generic unmarks) and then turns them off so normal matching will occur.

`^unmark()` will restore the set of generic unmarks that were flipped off using `^mark()`.

`^replaceword(word _0)`

You can change the word itself at the location just by providing the word you want used and the location in the sentence (as a match variable). Replacing a word does not make it visible to pattern matching. It is merely what will be retrieved (for both original and canonical).

`^position ( how matchvariable )`

This returns the integer representing where the named match variable is located.

how can be `START`, `END`, or `BOTH`. Both means an encoding of where the start and end of the the match was. See `@_n` in pattern matching to set a position or the `^setposition` function.

`^marked ( word )`

returns `1` if word is marked, returns `FAILRULE_BIT` if the given word is not currently marked from the current sentence.

`^setposition ( _var start end )`

Sets the match location data of a match var to the number values given.

Alternatively you can do `^setposition ( _var _var1 )`, which is redundant with just doing `_var = _var1`.

`^setcanon ( wordindex value )`

Changes the canonical value for this word.

`^settag ( wordindex value )`

Changes the pos tag for the word.

`^setoriginal ( wordindex value )`

Changes the original value for this word.

`^setrole ( wordindex value )`

Changes the parse role for this word. These are used in conjunction with `$cs_externaltag` to replace the CS inbuilt English postagger and parser with one from outside. See end of ChatScript PosParser manual.

`^savesentence ( label ) / ^restoresentence ( label )`

These two functions save and restore the current entire sentence preparation context. That means everything that pattern matching depends upon from the current sentence can be saved, you can go on to a new sentence (either via `^next(INPUT)` or `^analyze()` or whatever), and then rapidly flip back to some previous sentence analysis. Label is a value used to label the saved analysis. This only works during the current volley.

Cannot be used in document mode. `^savesentence` returns the number of 4-byte words the save took.

Input Functions

`^analyze ( stream )`

The stream generates output (not printed to user) and then prepares the content as though it were current input sentence. This means the current sentence flagging and marking are all replaced by this one's. It does not affect any pending input still to be processed. If the stream is quoted string, the quotes are removed. This would be common, for example, when analyzing output from the chatbot gotten via grabbing facts with "chatoutput" as the verb.

Note that the stream is considered a single sentence. If you want to supply multiple sentences, you need to call `^tokenize` and then loop on the facts created.

Note that `^analyze` does not call any prepass topic you may have, but you can invoke that topic directly aterwards yourself.

`^tokenize ( {WORD SENTENCE} stream )`

`WORD` or `SENTENCE` are optional parameters (`SENTENCE` is default).

If `SENTENCE`, then splits the stream into sentences and creates facts of each like this: `(sentence ^tokenize ^tokenize)`.

If `WORD`, then splits it entirely into words paying no attention to sentence boundaries.

`^capitalized ( n )`

Returns `1` if the nth word of the sentences starts with a capital letter in user input, else returns `0`.

If *n* is alphabetic, it returns whether or not it starts with a capital letter. Illegal values of n return failrule.

`^input ( ... )`

The arguments, separated by spaces, are injected back into the input stream as the next input, processed before any pending additional input. Typically this command is then followed by `^fail(SENTENCE)` to cancel current processing and move onto the revised input.

Since the sentence is fed in immediately after the current input, if you want to feed in multiple sentences, you must reverse the order so the last sentence to be processed is submitted via input first. You can detect that the current sentence comes from `^input` and not from the user by `%revisedInput` (bool) being true (1).

Note: the input sentence is not what the user originally typed, so don't expect it to reflect appropriately in %originalInput. Also it is tokenized on entry, so things like commas may already have been separated.

`^original ( _n )`

The argument is the name of a match variable. Whatever it has memorized will be used to locate the corresponding series of words in the original raw input from the user that led to this match. That is, the value is prior to any spell correction done by ChatScript.

```
u: (my _life) ^original(_0)
```

For input "my lif" spell correction will change the input to "life", which matches here, but ^original will return "lif".

`^position ( which _var )`

If *which* is `start` this returns the starting index of the word matched in the named `_var`.

If *which* is `end` this returns the ending index. E.g.,

if the value of `_1` was *the fox*, it might be that start was 3 and end was 4 in the sentence *it was the fox* .

^removetokenflags (value)

Rremoves these flags from the tokenflags returned from the preprocessing stage.

^settokenflags (value)

Adds these flags to the tokenflags return from the preprocessing stage. Particularly useful for setting the `#QUESTIONMARK` flag indicating the input was perceived to be a question.

For example, I treat *tell me about cars* sentences as questions by marking them as such from script (equivalent to *what do you know about cars?*).

Note that the change will not impact rule matching within the topic you have just done the change, because it has commited the set of rules it will try to match. So it only applies to later topics.

^setwildcardindex (value)

Tells the system to start at `value` for future allocations of wildcard slots. This is only useful inside some pattern where you are trying to protect data from some previous match. Eg.

```
u: (_~animals) refine()
    a: ( ^setwildcardindex(_1) _~color)
```

`_0` is set to an animal. Normally the rejoinder would set a color onto `_0` and clobber it, but the call to `^setwildcardindex` forces it to use `_1` instead, so both `_0` and `_1` have values.

^isnormalword (value)

Fails if value has a character that is not alphabetic, numberic, a hyphen, an underscore, or an apostrophe.

Number Functions

^compute (number operator number)

Performs arithmetic and puts the result into the output stream.

Numbers can be integer or float and will convert appropriately. There are a range of operators that have synonyms, so you can pass in directly what the

user wrote. The answer will be `?` if the operation makes no sense and infinity if you divide by `0`.

`~numberOperator` recognizes these operations:

operator symbol	description
`+`	plus add and (addition)
`-`	minus subtract deduct (subtraction)
`*`	x time multiply (multiplication)
`/`	divide quotient (float division)
`%`	remainder modulo mod (integer only- modulo)
`root`	square_root (square root)
`^^`	power exponent (exponent)
`<<` and `>>`	shift (limited to shifting 31 bits or less)
`random`	(0 random 7 means 0,1,2,3,4,5,6 - integer only)

Basic operations can be done directly in assignment statements like:

```
$var = $x + 43 - 28
```

`^timefromseconds ( seconds {offset} )`

This converts time in seconds (Unix epoch time) from the given time in whatever timezone, to a string like `%time` returns. You can compute a difference in times by merely doing a subtraction of the two times. `%fulltime` will give you the current time that you could plug in here. The optional second argument will displace that time by the hours offset (can be plus or minus).

`^timeinfofromseconds ( seconds )`

This converts time in seconds (Unix epoch time) into its component bits, spread across 7 match variables. Starting by default at `_0`, if you assign it like this:

```
_3 = ^timeinfofromseconds(%fulltime)
```

it will start at `_3`. The items you get are:

value	offset
seconds	0
minutes	1
hours	2
date in month	3
month name	4
year	5

value	offset
day name of week	6
month index (jan==0)	7
dayofweek index (sun==0)	8

^timetoseconds (seconds minutes hours date-of-month month year)

This converts time data since 1970 (Unix epoch time). Analogous to `%fulltime`, which returns the current time in seconds. Month can be number 1-12 or name of month or abbreviation of month. Date-of-month must be 1 or more. Year must be on or 1970 and less than 2100. Optional 7th argument indicates whether time is within daylight savings or not , values can be 1 or 0, t or f, T or F. Default is false.

^isnumber (value)

Fails if value is not an integer, float, or currency,

Output Functions

The following functions cannot be used during postprocessing since output has been finished in theory and you can now analyze it.

^flushoutput()

Takes any current pending output stream data and sends it out. If the rule later fails, the output has been protected and will still go out (though the rule will not erase itself).

^insertprint (where stream)

The stream will be put into output, but it will be placed before output number where or before output issued by the topic named by where. The output is safe in that even if the rule later fails, this output will go out. Before the where, you may put in output control flags as either a simple value or a value list in parens.

^keephistory (who count)

The history of either BOT or USER (values of who) will be cut back to the count give. This affects detecting repeated input on the part of the user or detecting repeating output by the chatbot.

^lastsaid ()

Returns what the bot said last volley.

^print (stream)

Sends the results of outputing that stream to the user. It is isolated from the normal output stream, and goes to the user whether or not one later generates a failure code from the rule. Before the output you may put in output control flags as either a simple value without a # (e.g., OUTPUT_EVALCODE) or a value list in parens.

Flags include:

Flag	description
OUTPUT_EVALCODE	is automatic, so not particularly useful. Useful ones would control how print decides to space things
OUTPUT_RAW	does not attempt to interpret (or { or [or "
OUTPUT_RETURNVALUE_ONLY	does not go to the user, is merely return as an answer. Print normally stores directly into the response system, meaning failing the rule later has no effect. Print normally does not return a value so you can't store it into a variable. And print has a number of flags that can affect its formatting that dont exist with normal output. This flag converts print into an ordinary function returning a value, reversing all those differences
OUTPUT_NOCOMMANUMBER	dont add commas to numbers
OUTPUT_NOQUOTES	remove quotes from strings
OUTPUT_NOUNDERSCORE	convert underscores to blanks

These flags apply to output as it is sent to the user:

Flag	description
RESPONSE_NONE	turn off all default response conversions
RESPONSE_UPPERSTART	force 1st character of output to be uppercase
RESPONSE_REMOVESPACEBEFORECOMMA	as the name says
RESPONSE_ALTERUNDERSCORES	convert underscores to spaces
RESPONSE_REMOVETILDE	remove leading ~ on class names
RESPONSE_NOCONVERTSPECIAL	don't convert ecaped n, r, and t into ascii direct characters
RESPONSE_CURLYQUOTES	change simple quotes to curly quotes (starting and ending)
RESPONSE_NOFACTUALIZE	suppresses building bot output facts (for postprocessing)

^preprint (stream)

The stream will be put into output, but it will be placed before all previously generated outputs instead of after, which is what usually happens. The output is safe in that even if the rule later fails, this output will go out. Before the output you may put in output control flags as either a simple value or a value list in parens.

^repeat ()

Allows this rule to generate output that may repeat what has been said recently by the chatbot.

^reviseOutput (n value)

Allows you to replace a generated response with the given value.

n is one based and must be within range of given responses. One can use this, for example, alter output to create accents. Using **^response** to get an output, you can then use **^substitute** to generate a revised one and put it back using this function.

Output Access

These functions allow you to find out what the chatbot has said and why.

`^response ( id )`

What the chatbot said for this response. *Id* `1` will be the first output.

`^responsequestion ( id )`

Boolean 1 if response ended in `?`, null otherwise.

`^responseruleid ( id )`

The rule tag generating this response from which you can get the topic. May be joined pair of rule tags if rule was relayed (reuse) from a different rule. The final rule will be first and the relay second, eg `~keywordless.30.0.~control.3.4`.

If the *id* is `-1`, then all output generated will be included, analogous to what happens in the log file for `why` in the entries.

`^responsepattern(responseid)`

Returns the pattern that matched inside [] of rule generating output (if it is matched that way) For a rule like: u: ([(pattern 1) (pattern 2) ([try 3])]) It will tell you which piece of the pattern matched. Handy for debugging why a pattern matches incorrectly w/o having to read a trace log and analyzing each + and -.

PostProcessing Functions

These functions are only available during postprocessing.

`^postprintbefore ( stream )`

It prints the stream prepended to the existing output. You will not be able to analyze or retrieve information about this, like you would from a normal print because it generates no facts representing it. This is useful for adding outofband messages `[ ]` to the front of input for controlling avatars and such. Or for adding transitional phrases or other personality coloring before the main output.

`^postprintafter ( stream )`

It prints the stream appended to the existing output. You will not be able to analyze or retrieve information about this, like you would from a normal print because it generates no facts representing it. This is useful for adding summarizing data after output, e.g., when running the document reader.

Control Flow Functions

`^argument ( n )`

Retrieves the nth argument of the calling outputmacro (1-based).

`^argument ( n ^fn )`

Looks backward in the callstack for the named outputmacro, and if found returns the nth argument passed to it. Failure will be reported for n out of range or `^fn` not in the call path.

This is an alterative access to function variable arguments, useful in a loop instead of having to access by variable name.

If *n* is `0`, the system merely tests whether the caller exists and fails if the caller is not in the path of this call.

‘^Bug(msg)

Generates a run-time bug. If you are compiling script at the time, it generates a std script bug trap. If you are not, it forces an error message into the bug log.

`^callstack ( @n )`

Generates a list of transient facts into the named factset. The facts represent the callstack and have as subject the critical value (the verb is `callstack` and the object is the rule tag responsible for this entry). Items include function calls (^xxxx) and topic calls (~xxxx) and internal calls (no prefix).

`^command ( args )`

Execute this stream of arguments through the `:` command processor. You can execute debugging commands through here. E.g.,

```
^command(:execute ^print("Hello") )
```

Note that it is hard to turn on :trace this way, because the system resets it internally at various points. The correct way to manipulate trace is to do `$cs_trace = -1` in regular script, outside of ^command.

`^end ( code )`

Takes 1 argument and returns a code which will stop processing. Any data pending in the output stream will be shipped to the user. If `^end` is contained within the condition of an if, it merely stops it. An end rule inside a loop merely stops the loop. All other codes propagate past the loop. The codes are:

code	description
`CALL`	stops the current outputmacro w/o failing it. See also ^return
`RULE`	stops the current rule. Whether the next rule triggers depends upon whether or not output was generated
`LOOP`	stops the current loop but not the rule containing it. Can pass up through topics to find the loop. If there is no loop, it will fail you all the way to the top
`TOPIC`	stops the current topic
`SENTENCE`	stops the current rule, topic, and sentence

code	description
INPUT	stops all the way through all sentences of the current input
PLAN	succeeds a plan - (only usable within a plan)

^eval (flags stream)

To evaluate a stream as though it were output (like to assign a variable). Can be used to execute :commands from script as well.

Flags are optional and match the flag capabilities of ^print.

One common flag would be OUTPUT_NOQUOTES if you wanted to remove the enclosing "" from a user variable value. E.g.,

```
$$tmp = ^eval(OUTPUT_NOQUOTES $$arg1)
```

^eval is also particularly used with variables, when you know the value of a variable is itself a variable name and you want its actual value, e.g.

```
$nox = 1
$$tmp = join($ no x)
$$val = eval($$tmp) # $$val = 1
```

^fail (code)

Takes 1 argument and returns a failure code which will stop processing. How extensive that stop is, depends on the code. If ^fail is contained within the condition of an if, it merely stops that and not anything broader. A fail or end rule inside a loop merely stops the loop; other forms propagate past the loop. The failure codes are:

code	description
RULE	stops the current rule and cancels pending output

code	description
`LOOP`	stops a containing loop and fails the rule calling it. If you have no containing loop, this can crawl up through all enclosing topics and make no output
`TOPIC`	stops not only the current rule also the current topic and cancels pending output. Rule processing stops for the topic, but as it exits, it passes up to the caller a downgraded fail(rule), so the caller can just continue executing other rules
`SENTENCE`	stops the current rule, the current topic, and the current sentence and cancels pending output

code	description
`INPUT`	stops processing anything more from this user's volley. Does not cancel pending output. It's the same as END(INPUT)

Output that has been recorded via `^print`, `^preprint`, etc is never canceled. Only pending output.

`^load ( name )`

Normally CS takes all the data you have compiled as `:build 0` and `:build` whatever as layers 0 and 1, and loads them when CS starts up. They are then permanently resident. However, you can also compile files named `filesxxx2.txt` which will NOT be loaded automatically.

You can write a script that calls `^load`, naming the `xxx` part and they will be dynamically loaded, for that user only, and stay loaded for that user across all volleys until you call `^load` again. Calling load again with a different name will load that new name. Calling `^load(null)` will merely unload the dynamic layer previously loaded.

WARNING

It's erroneous (you get whatever happens to you), if you call `^load` from within topics you have loaded via `^load`.

`^clearmatch()`

This clears all match variables to empty.

`^match ( what )`

This does a pattern match using the contents of what (usually a variable reference). It fails if the match against current input fails. It operates on the current analyzed sentence which is usually the current input, but since you can call `^next(input)` or `^analyze()` it is whatever the current analysis data is.

```
if (%more AND ^match(^"(< ![~emocurse ~emothanks] ~interjections >)" ) )
    {FAIL(SENTENCE)}
```

or

```
$$newrule = GetRule(pattern $$newtag)
$$newtype = GetRule(type $$newtag)
if ($$newtype == $$type AND match($$newrule)) # we would match this rule
```

`^match` can also take a rule tag for what, in which case it uses the pattern of the rule given it. `^match` will normally take your pattern and compile it with the script compiler during execution.

If you have discarded the script compiler in your build, it will run your pattern directly and pray. In that case every token should be separated by a space: eg not this:

```
[my you]
```

but this

```
[ my you ]
```

and relational tests won't work so you can't do `_0>5` or `_0?` or things like that. If you know your pattern in advance, you can put it on a rule and then execute that since it will have been compiled. E.g.

```
s: TEST (some fancy pattern)
```

and later

```
^match(~mytopic.test)
```

You can also just say `^match(~someconcept)` and it will test the current input for that concept.

`$$csmatch_start` and `$$csmatch_end` are assigned to provide the range of words that `^match` used.

`^matches ()`

Returns a string of indices of words that matched the most recent pattern match. The indices are in order, so you can know the range of the match or the specific word indices that were seen. Currently matches only include the words/concepts that were matched, not things like

```
(sag*)
```

where the word is not fully named.

^nofail (code ... script ...)

The antithesis of ^fail(). It takes a code and and number of script elements, executes the script and removes all failure codes through the listed code.

This is important when calling ^respond and ^gambit from a control script. You would want a control script to pass along codes at the sentence level, but if the respond call generated a fail-rule return, you don't want that to stop all the code of a control script responder.

The nofail codes are:

code	description
RULE	a rule failure within the script does not propagate outside of nofail
LOOP	a loop failure or end within the script does not propagate outside of nofail
TOPIC	a topic or rule failure within the script does not propagate outside of nofail
SENTENCE	a topic or rule or sentence failure within the script does not propogate outside of nofail
INPUT	no failure propagates outside of the script

notnull (stream)

Execute the stream and if it returns no text value whatsoever, fail this code. The text value is not used anywhere, just tested for existence. Useful in IF conditions.

^norejoinder ()

Prevents this rule from assigning a rejoinder.

^notrace (...)

Suppresses normal tracing if if :trace all is on, for the duration of evaluation of the contents of the parens. It does not block explicit traces of functions or topics.

^trace (...)

Enables trace for the following, even if exterior is controlled by a ^notrace directive.

^return (...)

Evaluates it data and returns any output from the most recent calling output-macro. It is nominally equivalent to:

```
here is some outputting
^end(CALL)
```

My personal coding convention is to use ^return when the function is supposed to return a value to a caller who will assign it somewhere. And not to use it if the function is directly creating output to the user or is just being executed for side effects.

You can return the contents of a variable ^return($$myvar) or the name of a factset ^return(@19) or just some literal value ^return(test). Returning a factset just returns its name. But if you have

```
@0 = ^myfunc()
```

and ^myfunc returns a factset name, you have done the equivalent of

```
@0 = @19
```

which means copy the elements of set 19 into set 0.

Note that ^return() and ^return(null) are treated the same. An empty string is returned. This is similar to assigning a variable by saying $var = null which assigns the empty string.

^addcontext (topic label)

Sets a topic and context name for use by ^incontext.

The *label* doesn't have to corrrespond to any real label.

The *topic* can be a topic name or ~ meaning current topic.

^authorized ()

Use same authorizedIP.txt file and rules that debug commands use, to validate current user.

^clearcontext ()

Erases all context data (see ^addcontext).

^incontext (label)

This function is only used inside a pattern.

label can be a simple text label or a `topicname.textlabel`. The system tracks rule labels that generated output to the user or rules starting with the label `CX_` whether or not the rule generates output as long as it didn't fail during output.

`^inContext` will return how many volleys have happened since the referenced rule (normal return) if the label has output within the 5 prior volleys and will fail if not. It's like an extension of rejoinders. Rejoinders have a 1 volley context and must be placed immediately after a rule. This has a 5 volley context and are used in normal rule patterns.

```
u: (^incontext(PLAYTENNIS) why) because it was fun.
```

^stats (FACTS / DICT / TEXT)

FACTS: Returns how many free facts remain. DICT: Returns how many empty dictionary entries remain. TEXT: Returns how much space for both stack and heap remain.

External Access Functions

^environment (variablename)

Access environment variables of the operating system. E.g.

```
^environment(path)
```

^system (any number of arguments)

The arguments, separated by spaces, are passed as a text string to the operating system for execution as a command. The function always succeeds, returning the return code of the call. You can transfer data back and forth via files by using `^import` and `^export` of facts.

^popen (commandstring 'function)

The command string is a string to pass the os shell to execute. That will return output strings (some number of them) which will have any `\r` or `\n` changed to blanks and then the string stripped of leading and trailing blanks.

The string is then wrapped in double quotes so it looks like a standard ChatScript single argument string, and sent to the declared function, which must be an output macro or system function name, preceded by a quote.

The function can do whatever it wants. Any output it prints to the output buffer will be concatenated together to be the output from ChatScript. If you need a doublequote in the command string, use a backslash in front of each one. They will be removed prior to sending the command. E.g.,

```
outputmacro: ^myfunc(^arg)
^arg \n
topic: ~test( testing )
u: () popen( "dir *.* /on" '^myfunc)
```

output this:

```
Volume in drive C is OS
Volume Serial Number is 24CB-C5FC

Directory of C:ChatScript

06/15/2013 12:50 PM <DIR> .
06/15/2013 12:50 PM <DIR> ..
12/30/2010 02:50 PM 5 authorizedIP.txt
06/15/2013 12:19 PM 10,744 changes.txt
05/08/2013 03:29 PM <DIR> DICT
...( additional lines omitted)
49 File(s) 29,813,641 bytes
24 Dir(s) 566,354,685,952 bytes free
```

`'Function` can be null if you are not needing to look at output.

`^tcpopen ( kind url data 'function )`

Analogous in spirit to popen.

You name the `kind` of service (`POST`, `GET`), the `url` (not including `http://`) but including any subdirectory, the text string to send as data, and the quoted function in ChatScript you want to receive the answer.

The answer will be read as strings of text (newlines separate and are stripped off with carriage returns) and each string is passed in turn to your function which takes a single argument (that text).

`:trace TRACE_TCP`

can be enabled to log what happens during the call.

Likely you will prefer `^jsonopen` which can deal with more complex web communication scenarios and returns structured data so you don't have to write

script yourself to parse the text.

'function can be null if you are not needing to look at output.

The system will set $$tcpopen_error with error information if this function fails.

When you look at a webpage you often see it's url looking like this:

http://xml.weather.com/weather/local/4f33?cc=*&unit ="+vunit+"&dayf=7"

There are three components to it.

The host: xml.weather.com.

The service or directory: /weather/local/4f33.

The arguments: everything AFTER the ?.

The arguments are URLencoded, so spaces have been replaced by +, special characters will be converted to %xx hex numbers.

If there are multiple values, they will be separated by & and the left side of an = is the argument name and the right side is the value.

When you call ^tcpopen, normally you provide the host and service as a single argument (everything to the left of ?) and the data as another argument (everything to the right of ?).

Since ChatScript URL encodes, you don't. If you don't know the unencoded form of the data or you don't think CS will get it right, you can provide URL-encoded data yourself, in which case make your first argument either POSTU or GETU, meaning you are supplying url-encoded data so CS should not do anything to your arguments.

Below is sample code to find current conditions and temperature in san francisco if you have an api key to the service. It calls the service, gets back all the JSON formatted data from the request, and line by line passes it to ^myfunc.

This, in turn, calls a topic to hunt selectively for fragments and save them, and when all the fragments we want have been found, ^myfunc outputs a message and stops further processing by calling ^END(RULE).

Note that in this example there is no data to pass, everything is in the service named, so the data value is "".

```
outputmacro: ^myfunc (^value)

$$tmp = ^value
nofail(RULE respond(~tempinfo))

if ($$currentCondition AND $$currentTEMP)
    {
    print( It is $$currentCondition. )
```

```
    print(The temperature is $$currentTemp. )
    ^END(RULE)
    }

topic: ~tempinfo system repeat keep()

u: (!$$currentCondition)
   $$start = findtext($$tmp $$pattern1 0)
   $$findtext_start = findtext($$tmp ^"\"" $$start)
   $$currentCondition = extract($$tmp $$start $$findtext_start )

u: ($$currentCondition)
    $$start = findtext($$tmp $$pattern2 0)
    $$findtext_start = findtext($$tmp , $$start)
    $$currentTemp = extract($$tmp $$start $$findtext_start)

topic: ~INTRODUCTIONS repeat keep (~emogoodbye ~emohello ~emohowzit name )

t: ^keep() Ready. Type "weather" to see the data.

u: (weather)
    $$pattern1 = ^"\"weather\":\""
    $$pattern2 = ^"\"temp_f\":"
    if ( tcpopen(GET api.wunderground.com/api/yourkey/conditions/q/CA/San_Francisco.json ""
        { hi }
    else
        { $$tcpopen_error }
```

There is a subtlety in the `^myfunc` code in that it uses ^print to put out the result. Just writing:

```
if ($$currentCondition AND $$currentTEMP)
    {
    It is $$currentCondition.
    The temperature is $$currentTemp.
    ^END(RULE)
    }
```

will not work, because that output is being generated by the call to `^tcpopen`, which is in the test part of the if, so everything it does is purely for effect of testing a condition. The generated output is dicarded.

If you moved the output generation to the `{ }` of the `if`, things would be fine. E.g.,

```
if ( tcpopen(GET api.wunderground.com/api/yourkey/conditions/q/CA/San_Francisco.json "" '^my
    {
    It is $$currentCondition.
```

```
    The temperature is $$currentTemp.
    }
else { $$tcpopen_error }
```

Doing the output without using `^print` is my preferred style; it is easier to see what is going on for output if it is not hidden deep inside some if test.

`^export ( name from )`

From must be a fact set to export. Name is the file to write them to. An optional 3rd argument `append` means to add to the file at the end, rather than recreate the file from scratch.

Obviously, you must first have done something like `^query` to populate the fact set. Eg.

```
^query(direct_sv item label ? -1 ? @3)
^export(myfacts.txt @3)
```

If the name includes the substring "ltm", then the file will not be appendable, but will be encryptable and routes to databases if the filesystem has been overridden by Mongo, Postgres, or MySQL.

`^import ( name set erase transient )`

name is the file to read from. Set is where to put the read facts.

erase can be `erase` meaning delete the file after use or `keep` meaning leave the file alone.

transient can be `transient` meaning mark facts as temporary (to self erase at end of volley) or `permanent` meaning keep the facts as part of user data. Eg

```
^import(myfacts.txt @3).
```

If *set* is null, then facts are created but not stored into any fact-set and the subject of the first fact is returned as the answer (presumed to be a json structure).

If the name includes the substring "ltm", then the file will be decryptable and routes to databases if the filesystem has been overridden by Mongo, Postgres, or MySQL.

CS External API- ^CompilePattern

The external API functions allow execution of rule behaviors from outside of ChatScript. This function takes in a pattern string like

```
(   [ ~bottle test] *  me)
```

and compiles it to internal format. Alternatively it accepts a std CS 'if' condition like

```
if ($var >= 1 AND !$var2)
```

It returns JSON object notation in ordinary user output, e.g.

```
{ "code": "(....)" }
```

You can take that code value and save it away somewhere and later pass it into ^testpattern. Should compilation fail, then there will be no CODE field and instead you will have an errors field.

```
{"errors": [ xxx yyy ] }
```

In either case you may see a warnings field

```
{"warnings": [ xxx yyy ] }
```

where xxx and yyy are warning messages.

The code returned may have additional data at the start. This is used to inform spelling correction to protect words occurring in the pattern. For best results you should compile patterns in the same bot environment you expect to execute them in. The pattern won't explictly protect words it thinks the bot will already be protecting.

CS External API- ^TestPattern

The external API functions allow execution of rule behaviors from outside of ChatScript. ^testpattern will execute a list of patterns and tell you which one (if any) matched and possibly return match variables. The argument is a JSON object as follows

```
{
"input": "range leak", -- required
"patterns": [  -- at least one required
    string-returned-from-compilepattern,
    string-returned-from-compilepattern,
    ],
"style": "earliest", -- optional
"trace": "1", -- optional
"variables": { -- optional
    "$faucet": "testing",
    "$x": 1 },
"trace": 1,     -- optional, returns a field "trace" which is CS trace pattern of the patte
"concepts": [  -- optional
    { "name": "~stove",
    "values": [ "oven", "range", "cooktop" ]
    },
```

```
    { "name": "~leaky",
    "values": [ "'leak", "drip", "spill" ] } ]
}
```

Input is the user input (one or more sentences) to be matched against.

Patterns is an array of pattern strings as returned by CompilePattern. They can use memorization and they can assign values. If they use memorization, you can assign those onto normal variables, which if permanent variables will be returned as "newglobals". You can perform an assignment inside the pattern using something like $answer:=_0 (see Match variable assignment in Advanced patterns). The "newglobals will be omitted if there are no changes.

The variables and concepts fields are optional and provide context. A concept named "$_{\text{replace}}$" is treated not as a concept but as a list of paired words analogous to "replace:", but is only a transient replace series for this call.

Style is also optional and defaults to `earliest`. Other choices are all, best, and latest. Earliest means stop running patterns and sentences as soon as you get a match. The makes the call faster and if your patterns are ordered best first, there is no incentive to try lower priority patterns or more sentences. Latest is the opposite, presumes that the patterns are ordered worst first. In addition, last will not execute any patterns earlier than the already matching one once one has been found (since it cannot improve the result). All runs all patterns on all sentences. It is good for gleaning data. Best presumes patterns are ordered best first, but once a match has been found it will move on to additional sentences to see if a better match can be found.

The optional `trace` field if set, will have the patterns involved traced (:trace pattern) and the results returned in a field named `trace` as a text string with crlf. Or a pattern can set the variable `$$cs_testpattern_style` to be a value and the system will enable trace until it is set off or the call completes.

Result is ordinary user output JSON object:

```
{
    "match": 1,  -- index of matching pattern
    "newglobals": { "$stovetype": "range", "$leaktype": "leak" }
}
```

If nothing matched, the value of `match` is false. Otherwise it is the index of the matching pattern (0-based). If there are return values from matching one or more patterns, those will be listed in `newglobals`, which is omitted if there are none. Values of null are never returned.

You can force ^testpattern to trace user regardless of whether tracing is on or not or whether nouserlog is set. Just prepend to your input ":tracepattern"

CS External API- ^CompileOutput

The external API functions allow execution of rule behaviors from outside of ChatScript. This function takes in an output string like

```
if ($test) {test me}
^callmycode($test)
```

and compiles it to internal format. It returns JSON object notation in ordinary user output, e.g.

```
{ "code": "(....)" }
```

You can take that code value and save it away somewhere and later pass it into ^testpattern. Should compilation fail, then there will be no CODE field and instead you will have an errors field.

```
{"errors": [ xxx yyy ] }
```

In either case you may see a warnings field

```
{"warnings": [ xxx yyy ] }
```

where xxx and yyy are warning messages.

CS External API- ^TestOutput

The external API functions allow execution of rule behaviors from outside of ChatScript. The argument is a JSON object as follows

```
{
"output":  code-returned-from-compile-output,
"variables": { -- optional
    "$faucet": "testing",
    "$x": 1 },
}
```

If a variable value above looks like a JSON arry or object, will be transformed into the corresponding internal JSON name.

The result is a JSON object

```
{
    output: "message to user",
    newglobals:  {"$x": "1", "$y": "testing"}  -- optional
}
```

Output may be blank if nothing is intended to be generated or the code fails in execution. Newglobals will be present if the code changes permanent global variables. Changes to transient variables $$ and local variables $_ will not be returned. If nothing is returned, the field is omitted. If a permanent variable

is set to a JSON structure, the entire JSON string will BE send (and not the JSON name).

Debugging Function `^debug ()`

As a last ditch, you can add this function call into a pattern or the output and it will call DebugCode in `functionExecute.cpp` so you know exactly where you are and can use a debugger to follow code thereafter if you can debug c code.

Logging Function `^log ( ... )`

This allows you to print something directly to the users log file. If you want it echoed to the console as well, you can do `^log(OUTPUT_ECHO This is my message)`.

You can actually append to any file by putting at the front of your output the word FILE in capital letters followed by the name of the file. E.g.,

```
^log(FILE TMP/mylog.txt This is my log output.)
```

Logging appends to the file. If you want to clear it first, issue a log command like this:

```
^log(FILE TMP/mylog.txt NEW This is my log output)
```

The `new` tells it to initialize the file to empty.

Additionally you can optimize log file behavior. If you expect to write to a file a lot during a volley (eg during :document mode), you can leave the file open by using

```
^log(OPEN TMP/mylog.txt This is my log output.)
```

which caches the file ptr. After which you can write with OPEN or FILE equivalently. To close the file use

```
^log(CLOSE TMP/mylog.txt)
```

By default, ^log acts like output to user, converting escaped n, r, and t into their actual ascii characters. The flag RESPONSE_NOCONVERTSPECIAL passed in will block this. You pass various flags as:

```
^logcode( flag OPEN ...)  or
^logcode( (flag flag) OPEN ...)
```

`^memorymark ()`

Reading a document consists of performing a single volley of the entire document. This can tie up a lot of memory in keeping facts, dictionary entries, user variables,

etc. If you are careful in what you do, you can make the memory burden go away. `^memoryMark()` notes where memory is currently at, and is best done within the document_pre topic. Then you can release memory after every sentence of the document, so it doesn't accumulate.

Multiple calls to memory mark stack up the data, so each call to ^memoryfree will undo one mark.

`^memoryfree ( {$data})`

This releases memory back to the most recent ^memorymark()‘. It is best done after your main control of the document bot has finished processing a sentence. Partly because the analysis of the sentence is lost and so no later rules can pattern match to it (though you can call ^analyze to reacquire your sentence). E.g.,

```
topic: ~document_pre system repeat()

t: ^memorymark() # note start
  Log(OUTPUT_ECHO \n Begin $$document ) # instant display

topic: ~main_control system repeat () # executed each sentence of document

u: (%document)
  respond(~filter)
  ^memoryfree()
```

The caveats and warnings about how this works. Whenenver you free memory, the system will clear all fact sets. It will clear all user variables set after the memory mark (leaving the ones before alone). It will then release facts, text, and dictionary nodes created after the mark.

The only data you can pass out from a memoryMark/memoryfree zone is data stored on match variables (which have size limitations) or on the count field of a dictionary word of a preexisting word or in the argument (if supplied) of $data (hence a nice JSON structure you can pass).

`^memorygc ()`

This can function in either document mode or chat mode. It does what it can to release unused memory. It has restrictions in it does not work if you have facts with facts as fields or are in planning mode. It also discards saved sentence data, and all of your analysis data for the current sentence. It also discards all data in factsets.

JSON Functions

JSON functions and JSON are described more fully in the ChatScript JSON manual.

^jsonarrayinsert (arrayname value)

Given the name of a json array and a value, it adds the value to the end of the array. SAFE protects any nested JSON data from being deleted. See JSON manual.

^jsonarraydelete ([INDEX, VALUE] arrayname value {ALL})

This deletes a single entry from a JSON array. It does not damage the thing deleted, just its member in the array. If the first argument is INDEX, then value is a number which is the array index (0 ... n-1). If the first argument is VALUE, then value is the value to find and remove as the object of the json fact.

You can delete every matching VALUE entry by adding the optional 4th argument ALL.

If there are numbered elements after this one, then those elements immediately renumber downwards so that the array indexing range is contiguous.

^jsoncreate (type)

Type is either array or object and a json composite with no content is created and its name returned.

^jsondelete (factid)

Deprecated in favor of ^delete.

^jsongather ({factset} jsonid)

Takes the facts involved in the json data (as returned by ^jsonparse or ^jsonopen and stores them in the named factset. This allows you to remove their transient flags or save them in the users permanent data file.

You can omit fact-set as an argument if you are using an assignment statement:

```
@1 = ^jsongather(jsonid)
```

`^Jsongather` normally gathers all levels of the data recursively. You can limit how far down it goes by supplying `level`. Level 0 is all. Level 1 is the top level of data. Etc.

`^jsonlabel ( label )`

Assigns a text sequence to add to `jo-` and `ja-` items created thereafter. E.g. `^jsonlabel(x)` generates `jo-x1` and `ja-x1`. You can turn it back off again with `^jsonlabel("")`

This allows you to create json namespaces which will not conflict. Eg, you may load a bunch of json during a system bootup (`^csboot`) under one naming and then use a different naming for user json created later and code can determine the source of the data.

`^readfile ( TAB filepath )`

`^readfile ( TAB filepath )` reads a tsv (tab delimited spreadsheet file) and returns a JSON array representing it. The lines are all objects in an array. The line is an object where non-empty fields are given as field indexes. The first field is 0. Empty fields are skipped over and their number omitted. `^readfile ( TAB filepath 'function)` reads tsv and spreads fields onto arguments of 'function, which it calls onceper line `^readfile ( LINE filepath 'function)` reads any file and passes each line untouched as the sole argument to the function.

Fomerly called ^jsonreadcsv.

`^jsonundecodestring ( string )`

Removes all json escape markers back to normal for possible printout to a user. This translates `\n` to newline, `\r` to carriage return, `\t` to tab, and `\"` to a simple quote.

`^jsonobjectinsert ( {DUPLICATE} objectname key value )`

Inserts the key value pair into the object named. The key does not require quoting. Inserting a json string as value requires a quoted string. Duplicate keys are ignored unless the optional 1st argument `DUPLICATE` is given. `SAFE` protects any nested JSON data from being deleted. See JSON manual.

^jsonopen ({UNIQUE} kind url postdata header)

This function queries a website and returns a JSON datastructure as facts. It uses the standard CURL library, so it's arguments and how to use them are generally defined by CURL documentation and the website you intend to access. See ChatScript JSON manual for details.

^jsontree (name)

`name` is the value returned by `^JSONparse`, `^JSONopen`, or some query into such structures. It prints out a tree of elements, one per line, where depth is represented as more deeply indented. Objects are marked with { } as they are in JSON. Arrays are marked with `[]`.

^jsonwrite (name)

`name` is the name from a json fact set (returned by `^JSONparse`, `^JSONopen`, or some query into such structures). Result is the corresponding JSON string (as a website might emit), without any linefeeds.

^jsonparse ({UNIQUE} string)

`string` is a json text string (as might be returned from a website) and this parses into facts exactly as `^jsonopen` would do, just not retrieving the string from the web. It returns the name of the root node. One use for this is to pass JSON data as a quoted string within out-of-band data, and have the system parse that into facts you can use.

You can add `NOFAIL` before the string argument, to tell it to return null but not fail if a dereference path fails cannot be found.

```
^jsonparse(transient NOFAIL "{ a: $var, b: _0.e[2] }")
```

`^jsonparse` automatically converts any backslashunnnn into the corresponding utf8 character.

^jsonkind (something)

If *something* is a JSON object, the function returns `object`. If it is a JSON array it returns `array`. Otherwise it fails.

```
^jsonpath ( string id )
```

string is a description of how to walk JSON. Id is the name of the node you want to start at (typically returned from `^jsonopen` or `^jsonparse`.

Array values are accessed using typical array notation like `ja-1[3]` and object fields using dotted notation like `jo-7.id`.

A simple path access might look like this: `[1].id` which means take the root object passed as id, e.g., `ja-1`, get the 2nd index value (arrays are 0-based in JSON). That value is expected to be an object, so return the value corresponding to the id field of that object. In more complex situations, the value of id might itself be an object or an array, which you could continue indexing like `[1].id.firstname`.

`^Jsonpath` can also return the actual factid of the match, instead of the object of the fact. This would allow you to see the index of a found array element, or the json object/array name involved. Or you could use `^revisefact` to change the specific value of that fact (not creating a new fact). Just add `*` after your final path, eg

```
^jsonpath(.name* $$obj)
^jsonpath(.name[4]* $$obj)
```

If you need to handle the full range of legal keys in json, you can use text string notation like this

```
^jsonpath(."st. helen".data $tmp)
```

You may omit the leading . of a path and CS will by default assume it

```
^jsonpath("st. helen".data $tmp)
```

Word Manipulation Functions

```
^burst ( {count once} data-source burst-character-string )
```

Takes the data source text and hunts within it for instances of the burst-character-string. If it is being dumped to the output stream then only the first piece is dumped.

If it is being assigned to a fact set (like `@2`) then a series of transient facts are created for the pieces, with the piece as the subject and `^burst ^burst` as the verb and object.

If it is being assigned to a match variable, then pieces are assigned starting at that variable and moving on to successively higher ones.

If burst does not find a separator, it puts out the original value. For assignment to match variables, it also clears the next match variable so the end of the list will be a null match variable.

If burst_character is omitted, it is presumed to be BOTH `_` (which joins composite words and names) and " ", which separates words.

If burst_character is the null string "", it means burst into characters.

`^burst` takes an optional first parameter `count`, which tells it to return how many items it would return if you burst, but not to do the burst.
`^burst` takes an optional first parameter `once` which says split only into the first burst and then the leftover rest.

`^burst` has a special burst value `digitsplit` which will split a number-text thing or a text-number thing into two pieces (text thing and number thing). This is good for splitting a currency thing lik USD25 or 25$.

`^words ( someword )`

Looks up the given word and returns all words matching it. Matching includes the lower case form of it and any number of uppercase forms of it. E.g, you might say `^words(ted)` and get back facts for *ted*, *Ted*, *TED*.

The answers are a series of facts of the form (someword words words). In addition to case switching, the system will automatically switch words with underscores or blanks into words with changes in them to the other (since CS stores phrases with underscores). So `^words("I love you")` can match phrases already in the dictionary of: *I_love you I_love_you I love you I LOVE You*

etc. Depending on which words are actually there (for example because they are parts of a fact).

`^canon ( word canonicalform )`

Same as `:canon` during a `:build` from a table. Fails during normal execution not involving compiling.

`^explode ( word )`

Convert a word into a series of facts of its letters.

`^extract ( source start end )`

Return the substring with the designated offset range (exclusive of end location). Useful for data extraction using ^popen and ^tcpopen when combined with

`^findtext`.

In addition to absolute unsigned values, start and end can take on offsets or relative values. A signed end is a length to extract plus a direction or shift in start:

```
^extract($$source 5 +2) # to extract 2 characters beginning at position 5

^extract($$source 5 -2) # to extract 2 characters ending at position 5
```

A negative start is a backwards offset from end.

```
^extract($$source -1 +1) # from end, 1 character before and get 1 character

^extract($$source -5 -1) # from end, 5 characters before and get 1 character before. i.e. th

^extract($$source -5 -1000) # all characters until 5 from end
```

Note start offset is 0-based indexing, but if you used ^findtext, it was 1-based index so you probably need to subtract 1.

`^findtext ( source substring offset {insensitive} )`

Find case sensitive substring within source+offset and return offset starting immediately after match. Useful for data extraction using `^popen` and `^tcpopen` when combined with `^extract`. `$$findtext_start` is bound to the actual start of the match. `$$findtext_word` is bound to the word index in which the match was found where one or more blanks separate words. Indexing starts at 1 (same as sentence positional notation).

An optional fourth argument `insensitive` will match insensitively.

Failing to match will generate a rule failure. If the source or substring contains an `_`, these will be converted to blanks before execution, to allow that or the space notation to be considered equivalent (unless your source or substring is literally an underscore only).

If you want to find a newline or tab character, then pass in the string `\n` or `\t`. That will find an actual ascii character of such. If you want to find the ascii string `\n`, then use \n and \t to find them. Normal scripts don't have ascii newline or tab in them. You wrote the backslashed characters and they are converted to the appropriate ascii characters on output to the user. But if you have read data from an external source, it will likely be the actual ascii characters.

`^flags ( word )`

get the 64bit systemflags of a word.

`^intersectwords ( arg1 arg2 optional )`

Given two "sentences", finds words in common in both of them. Output facts will go to the set assigned to, or `@0` if not an assignment statement. The optional third argument, if it's `canonical`, it will match the canonical forms of each word.

`^join ( any number of arguments )`

Concatenates them all together, putting the result into the output stream. If the first argument is `AUTOSPACE`, it will put a single space between each of the joined arguments automatically.

`^actualinputrange ( start end )`

Given the starting and ending word positions of an original input (what CS had after tokenization but before adjustments), this returns the range of where the words arose in the actual input. The return is a range whose start is shifted 8 bits left and ORed with the end position.

`^originalinputrange ( start end )`

Given the starting and ending word positions of an actual input (what CS sees after adjustments and what you normally pattern match on), this returns the range of where the words came from in the original input. The return is a range whose start is shifted 8 bits left and ORed with the end position.

`^properties ( word )`

Returns the 64bit properties of a word or fail-rule if the word is not already in the dictionary.

`^pos( part-of-speech word supplemental-data )`

Generates a particular form of a word in any form and puts it in the output stream. If it cannot generate the request, it issues a RULE failure. Most combinations of arguments are obvious. Here are the 1st & 3rd choices. For verbs with irregular pronoun conjugation, supply 4th argument of pronoun to use.

part-of-speech	word/verb/number(+ supplement-data argument)	action
`conjugate`	pos-integer(as returned from `^partofspeech`)	returns the word with that part of speech (eg conjugate go #VERB_PAST_PARTICIPLE)
`raw`	integer 1 .. `%length`	(returns the original word in sentence)
`syllable`	word	tells you how many syllables a word has
`hex64`	integer-word	converts a number to 64bit hex
`hex32`	integer-word	converts a number to 32 bit hex
`ismodelnumber`	word	return 1 if it is (mixed alpha/numeric). Fails otherwise.
`isinteger`	word	return 1 if it is all digits, fails otherwise
`isfloat`	word	return 1 if it is float, fails otherwise
`isuppercase`	word	return 1 if it begins with an uppercase letter, fails otherwise
`isalluppercase`	word	return 1 if it starts uppercase, and consists of entirely uppercase letters, hyphen, underscore and ampersand, fails otherwise
`ismixed case`	word	return 1 if it has both upper and lowercase letters, fails otherwise
`type`	word	returns concept, number, word, or unknown
`common`	word	returns level of commonness of the word
`verb`	verb	given verb in any form, return requested form
`present_participle`	verb	
`past_participle`	verb	
`infinitive`	verb	

part-of-speech	word/verb/number(+ supplement-data argument)	action
`past`	verb	
`present3ps`	verb	
`present`	verb	
`verb`	`match` noun	returns noun form matching verb (sing./plural).e.g. (*walk match dog*) -> *walks*
`aux`	auxverb `pronoun`	returns verb form matching pronoun supplied.for *do*,*have*, *be*
`pronoun`	word `flip`	changes person form for 1st and 2nd person
`adjective`	word `more`	writes the adjective in its comparative form: *fast* -> *faster*
`most`	word	the superlative form. *beautiful* -> *most beautiful*
`adverb`	word `more`	writes comparative form: *strong* -> *strongly*
`noun`	word `proper`	return word as a proper noun (appropriately cased)
`lowercaseexist`	word	
`uppercaseexist`	word	
`singular`	word or a number	== 1
`plural`	word or a number	> 1
`irregular`	word	return value only for irregular nouns
`determiner`	word `noun`	add a determiner "a/an" if it needs one
`place`	integer	return place number of integer
`capitalize`	word	
`uppercase`	word	
`lowercase`	word	
`allupper`	word	
`canonical`	word	see notes
`integer`	floatnumber	generate integer if float is exact integer

part-of-speech	word/verb/number(+ supplement-data argument)	action
`preexists`	word	return 1 if word in any casing was already in the dictionary before this volley, fail otherwise.

Example:

```
# get first name (in a not English language), and capitalize
u: what's your first name?
    #! giuditta
    a: ( _* )
   $_name = ^original(_0)
   Nice to meet you, ^pos(capitalize $_name)
   # if user enter giuditta, the rejoinder output: Nice to meet you, Giuditta
```

For ^pos(canonical), there is an optional third argument which is the concept name of the pos-tag. Foreign words may have multiple lemma forms based on part of speech. E.g., in the German dictionary you can find this entry:

```
Informationstechnische ( NOUN ADJECTIVE NOUN_SINGULAR NOUN_PLURAL ) lemma=`informationst
```

which says there are two forms of canonical, one for ADJA (adjective) and one for NN (noun). If you don't specify a 3rd argument, you get the first one (ADJA). If you specify `~ADJA` you get the first and if you specify `~NN` you get the second.

If your third argumernt is `all` then the list of all canonical forms is returned with | separating the entries.

`^decodeInputtoken ( number )`

Display the text values of tokenflag bits. You can pass it `%token` to see the meanings of the current sentence analysis or `$cs_token` to see what you have current set as token controls.

`^decodepos ( pos location )`

Translates into text the 64bit pos data at given location. `location` can be a position in the sentence (1... number of words) or a match variable found from some location in the sentence). See *dictionary.h* for meanings of bits. Type word will classify word as concept, word, number, or unknown.

`^decodepos ( role location )`

Returns the text of the role data of the given location.

`^layer ( word )`

When was this word entered into the dictionary. Answers are: `wordnet`, `0`, `1`, `2`, `user`.

`^partofspeech ( location )`

Gets the 64-bit part-of-speech information about a word at `location`, resulting from parsing. Location can be a position in the sentence (1... number of words) or a match variable found from some location in the sentence). See *dictionary.h* for meanings of bits.

`^phrase ( type matchvar )`

Can be used to retrieve all of a prepositional phrase or a noun phrase. `type` is `noun`, `prepositional`, `verbal`, `adjective`. Optional 3rd argument `canonical` will return the canonical phrase rather than the original phrase. E.g.,for input:

```
u: (I ~verb _~directobject) $tmp = ^phrase(noun _0)
```

with input *I love red herring* `$tmp` is set to *red herring*

`^role ( location )`

Gets the 32-bit role information about a word at location, resulting from parsing. Location can be a position in the sentence (1... number of words) or a match variable found from some location in the sentence). See *dictionary.h* for meanings of bits.

`^tally ( word {value} )`

Only valid during current volley. You can associate a 32-bit number with a word by `^tally(test 35)` and retrieve it via `^tally(test)`.

`^rhyme ( word )`

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

^substitute (mode find oldtext newtext)

Outputs the result of substitution. Mode can be character or word or insensitive. In the text given by find, the system will search for oldtext and replace it with newtext, for all occurrences. This is non-recursive, so it does not also substitute within replaced text. Since find is a single argument, you pass a phrase or sentence by using underscores instead of spaces. `^substitute` will convert all underscores to spaces before beginning substitution and will output the spaced results.

In character mode, the system finds oldtext as characters anywhere in newtext. In word mode it only finds it as whole words in newtext. Finding is case sensitive, unless you use the argument insensitive, which will do character mode insensitive match. You can select insensitive word match by making the first argument be a text string containing the normal 1st argument values, e.g. *insensitive word*

```
^substitute(word "I love lovely flowers" love hate)
```

outputs *I hate lovely flowers*

```
^substitute(character "I love lovely flowers" love hate)
```

outputs *I hate hately flowers*

^spell (pattern fact-set)

Given a pattern, find words from the dictionary that meets it and create facts for them that get stored in the referenced fact set. The facts are created with subject 1, verb word, and object the found word. The pattern is a text string describing possibly the length and letter constraints.

If there is an exact length of word, it must be first in the pattern. After which the system matches the letters you provide against the start of the word up until your pattern either ends or has an asterisk or a period. A period means match any letter.

An asterisk matches any number of letters and would normally be followed by more letters. The `*` will swallow letters in the dictionary word until it can match the rest of your given pattern. It will keep trying as needed. Eg.

```
^spell(4the @1) will find them but not their
^spell(am*ic @1) will find American
^spell(a*ent @1) will find abasement
^spell(h.l.o @1) will find hello
```

^sexed (word he-choice she-choice it-choice)

Given a word, depending on its sex the system outputs one of the three sex choices given. An unrecognized word uses it.

```
^sexed(Georgina he she it)
```

would return *she*

^uppercase (word)

Is the given word starting with an uppercase letter? Match variable binds usually reflect how the user entered the word. This allows you to see what case they entered it in. Returns 1 if yes and 0 otherwise.

^format(formatstring value)

This is a thin wrapper over sprintf. The first argument is a string which is the format string for sprintf. The second argument is the number to convert. For integer, if you use a %d format, you will be using a 32-bit value. For ll formats you will be using 64-bit but it won't work well on Windows output because Windows uses their own sprintf notation.

^addproperty (word flag1 ... flagn)

given the word, the dictionary entry for it is marked with additional properties, the flags given which must match property flags or system flags in *dictionarySystem.h*. Typically used to mark up titles of books and things when building world data.

In particular, however, if you are adding phrases or words not in the dictionary which will be used as patterns in match, you should mark them with `PATTERN_WORD`. To create a dynamic concept, mark the set name as CONCEPT.

You can also add fact properties to all members of a set of facts via

```
^addproperty(@4 flag1 ... flagn).
```

These flags are also predefined in dictionarysystem.h and you can use some of the predefined but meaningless ones to do what you want. These are `User_flag4`, `User_flag3`, `User_flag2`, `User_flag1`.

^define(word)

Output the definition of the word. An optional second argument is the part of speech: noun verb adjective adverb, which will limit the definition to just that

part of speech. Never fails but may return null.

The second argument can also be `all` which means list all definitions per part of speech, not just the first. And it can be the third optional argument so you can get all meanings of a word as a noun, for example.

`^hasanyproperty ( word value )`

Does this word have any of these property or systemflag bits? You can have up to 5 values as arguments, e.g.,

`^hasproperty(dog NOUN VERB ADJECTIVE ADVERB PREPOSITION)`

If the word is not in the dictionary, it will infer it, allowing it to handle things like verb tenses. If you want to insure the word already exists first, you should do ^properties(dog) AND `^hasproperty(dog xxx)` since property fails if the word is not found.

`^hasallproperty( word value )`

Does this word have all property or systemflag bits mentioned? You can have up to 5 values as arguments, e.g.,

`^hasallproperties(dog NOUN VERB ADJECTIVE ADVERB PREPOSTION)`

Values should be all upper case. If the word is not in the dictionary, it will infer it, allowing it to handle things like verb tenses. If you want to insure the word already exists first, you should do

`^properties(dog) AND ^hasproperty(dog xxx)`

since property fails if the word is not found.

`^removeinternalflag ( word value )`

Removes named internal flag from word.

Currently only value is `HAS_SUBSTITUTE`, which allows you to disable a word/phrase substitution. Use as word the full text of the left entry in a substitutions file. E.g.,

`<constantly>` maps to `~yes` normally. If you do `^removeinternalflag( <constantly> HAS_SUBSTITUTE)` then it will no longer do that.

This is a permanent change to the resident dictionary, which will take effect until the system is reloaded.

`^removeproperty ( word value )`

Remove this property bit from this word.

This effect lasts until the system is reloaded. Value should be all upper case. Value is normally a system flag value or a property value from `dictionarysystem.h` which does not need a hash in front of it (system will look up the name).

word can be in doublequotes. And there are two internal bits that are also allowed to be removed: `CONCEPT` and `HAS_SUBSITUTE`.

You can use `HAS_SUBSTITUTE` to disable some standard substitution in LIVEDATA, but you can't apply this at build time because the system won't remember. Instead call it from `^csboot` during startup. For example, in LIVEDATA interjections file, there is an entry:

```
<surprise ~emosurprise
```

But if you didn't want surpise at the start of a sentence declared the interjection `~emosurprise`, you could do

```
^removeproperty("<surprise" HAS_SUBSTITUTE)
```

And the dictionary has some words which are composites, like *iced_coffee* and it will automatically convert the two words into a single token. If you wanted to stop this behavior, you could disable this composite word via

```
^removeproperty(iced_coffee NOUN)
```

since it is declared as a NOUN (you can see with `:word iced_coffee`). You can do this to nouns, adjectives, adverbs.

`^walkdictionary ( '^function )`

calls the named output macro from every word in the dictionary. The function should have 1 argument, the word.

`^walktopics ( '^function )`

calls the named output macro for every topic the current bot can access. The function should have 1 argument, the topic name.

`^walkvariables ( '^function )`

calls the named output macro for every global permanent user variable currently non-null. The function should have 1 argument, the topic name. It does not involve $$ or $_ variables, or bot variables.

```
^Iterator ( ? member ~concept )
```

An iterator is a repeatable fact query that allows you to walk through each member of a concept, either at top level or recursively. Useful in conjunction with a `loop()`, the function is defined in the planning manual but can be used outside of planning. You can have one iterator in progress per rule.

```
loop () # unload every resource on board
    {
    $$resource = ^iterator(? member ~resources)
    ...
   }
```

```
^wordAtIndex ( ({original, canonical} n))
```

`^wordAtIndex` retrieves the word from the current sentence at the index given, as either the original word or as a canonical word (as a match variable sees it)

`^wordAtIndex ( canonical n n1)` gathers a range from n thru n1.

`^wordAtIndex ( original "_0")` gathers a range from that which _0 represents (but uses the original data so it is not like merely saying _0, which may not have real data if you did an arbitrary assignment to it setting its position).

Multipurpose Functions

```
^disable ( what ? )
```

What can be `topic` or `rule` or `inputrejoinder` or `outputrejoinder` or `save` or `write @set`.

If `topic`, the next argument can be a topic name (with or without `~` or just `~` meaning the current topic). It means to disable (BLOCK) that topic.

If a `rule`, you erase (disable) the labeled rule (or rule tag and `~` means the current rule).

If `outputrejoinder`, it cancels the current output rejoinder mark, allowing a new rule to set a rejoinder.

If `inputrejoinder` then it cancels any pending rejoinder on input.

If `save`, then the user data will not be saved. It's as though this interchange didn't happen.

If `^disable(write @1)` then factset `@1` will not be written out into user data and is junk at the next volley (normally this is true).

You can also disable the `inputrejoinder` with `^setrejoinder(input)` and the output rejoinder with `^setrejoinder(output)`.

`^enable ( what ? )`

What can be `topic` or `rule` or `save` or `write @set`.

If `topic`, thenext argument can be a topic name or the word `all` for all topics . Designated topics will be enabld (unBlocked).

If a `rule`, the label (or rule tag) will be enabled, allowing the rule to function again. If save, then it reenables saving user context if you had disabled it.

If `^enable(write @1)` then factset `@1` will be written out into user data and is restored at the next volley (normally this is not true).

`^length ( what )`

If what is a fact set like `@1`, length returns how many facts are in the set. If what is a word, length counts its characters. If what is a concept set, length returns a count of the top level (nonrecursive) members. If the name of a json array or object, returns how many top level elements it has.

`^pick ( what )`

Retrieve a random member of the concept if what is a concept. Pick is also used with factsets to pick a random fact (see FACTS MANUAL).

For a concept, if the member chosen is itself a concept, the system will recurse to pick randomly from that concept. If the argument to pick is a $var or _var, it will be evaluated and then pick will be tried on the result (but it won't recurse to try that again).

If the argument is a JSON object it randomly picks a fact whose verb/object is a key-value pair. If the argument is a JSON array, it randomly picks a fact whose verb is the index and whose object is the value. The fact id returned can be used with ^field or you can use something like $result.object to get the specific object.

`^reset ( what ? )`

What can be user or topic or factset or VARIABLES or FACTS or HISTORY.

If what is user, the system drops all history and starts the user afresh from first meeting (launching a new conversation), having erased the user topic file.

If what is a factset, the "next" pointer for walking the set is reset back to the beginning. If what is a topic, all rules are re-enabled and all last accessed values are reset to 0. If what is VARIABLES then it sets all global user variables to NULL, leaving alone
$$, $_ and bot variables. If what is FACTS, it kills all permanent user facts If what is HISTORY it forgets what was said previously.

FACT FUNCTIONS

`^findfact ( subject verb object )`

The simplest fact find involves knowing all the components (meanings) and asking if the fact already exists. If it does, it returns the index of the fact. If it doesn't it returns `FAILRULE_BIT`.

`^query ( kind subject verb object)`

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

query kind	description
direct_s	find all facts with the given subject
direct_v	find all facts with the given verb
direct_o	find all facts with the given object
direct_sv	find all facts with the given subject and verb
direct_so	find all facts with the given subject and object
direct_vo	find all facts with the given object and verb
direct_svo	find all facts given all fields (prove that this fact exists)
Unipropogate	find how subject joins into the object set

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

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

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

```
^query( kind subject verb object count fromset toset propagate
match )
```

query can actually take up to 9 arguments. Default values are `?`.

The *count* argument defaults to `-1` and indicates how many answers to limit to. When you just want or expect a single one, use 1 as the value.

fromset specifies that the set of initial values should come from the designated factset.

Special values of fromset are `user` and `system` which do not name where the facts come from but specify that matching facts should only come from the named domain of facts.

toset names where to store the answers. Commonly you don't name it because you did an assignment like

```
@3 = ^query(...)
```

and if you didn't do that, toset defaults to `@0` so

```
if (^query(direct_s you ? ?))
```

puts its answers in `@0`. It is equivalent to:

```
if (^query(direct_s you ? ? -1 ? @0))
```

You can also simultaneously query and unpack a single field from the first matching fact by naming the field on the to argument.

```
$$tmp = ^query(direct_sv you eat ? 1 ? @5object)
```

Typically you would do this if you expected only a single fact and were trying to find a specific field value. So you'd set the limit to 1. If a fact is found, it is stored in `@5`, and the object field is pulled off and stored onto `$$tmp`. If no fact is found, the query does not fail, $$tmp is merely set to null.

The final two arguments only make sense with specific query types that use those arguments.

For `unipropogate`, if you have these concepts;

```
concept: ~things (~animals ~vegetables ~minerals)
concept: ~animals (~canine ~feline)
concept: ~canine (dog)
```

Then `^query(unipropogate dog ? ~things 1)` would return `(~animals member ~things)`.

Note that the set to be found (`~things`) is not expanded. Normal queries expand any reference to a set into all of its members and expand simple words to the entire wordnet hierarchy above it. You can block this expansion behavior by

putting a single quote in front. Note for the idiom `'_0` which means the original form of the match variable, you have to use two quotes: `''_0`.

```
^query(direct_svo 'bomb ''_0 '$$tmp)
```

`unipropogate` expects a set as its object argument, so it does not need to be quoted. A query can also be part of an assignment statement, in which case the destination set argument (if supplied) is ignored in favor of the left side of the assignment, and the query doesn't fail even if it finds no values. E.g.,

```
@2 = ^query(direct_sv I love you)
```

The above query will store its results (including no facts found) in `@2`. Queries can also be used as test conditions in patterns and if constructs. A query that finds nothing fails, so you can do:

```
u: ( dog ^query(direct_sv dog wants ?)) A dog wants @0object.
```

You can also do `!^query`. Or

```
if (^query(direct_vo ? want toy)) {@0subject wants a toy.}
```

`^first ( fact-set )`

Retrieve the first fact, e.g.

```
_1 = ^first(@1all)
```

`^last ( fact-set )`

Retrieve the last fact.

`^pick ( fact-set )`

Retrieve a random fact. Removing the fact is the default, but you can suppress it with the optional second argument `KEEP`, e.g., `_1 = ^last(@1all)` gets the last value but leaves it in the set.

You can erase the contents of a fact-set merely by assigning null into it.

```
@1 = null
```

This does not destroy the facts; merely the collection of them.

`^sort ( {alpha alphabetic age} @0 ... )`

You can sort a fact set which has number values as a field.

^sort (fact-set {more fact sets})

The fact set is sorted from highest first. By default, the subject is treated as a float for sorting. You can say something like `@2object` to sort on the object field. You can add additional factsets after the first, which will move their contents slaved to how the first one was rearranged. Eg.

`^sort(@1subject @2 @3)`

will perform the sort using the subject field of `@1`, and then rearrange `@2` and `@3` in the same way (assuming they have the same counts). Instead of sorting by numeric value, can do an alpha sort or an oldest fact first sort similar to the normal sort.

^delete (factset or factid or jsonid)

If you actually want to destroy facts, you can query them into a fact-set and then do this:

`^delete(@1)` And all facts in `@1` will be deleted and the set erased You can also delete an individual fact who's id is sitting on some variable

`^delete($$f)` And you can delete a json array or object, including all of its substructure the same way.

If you pass something that is not deleteable, the system will do nothing and does not fail.

^length (factset or~setor jsonid or word)

If you want to know how many facts a fact-set has, you can do this:

`^length(@1)` - outputs the count of facts

Likewise how many top level members of a concept (not recursive). Or how many fields in a json object or elements in a json array. Or how many characters in a word. A Null value for the argument is legal, and is of length 0.

Note: if you do length of a name that starts with `~` and is not a defined concept set, the function fails rather than return 0.

^nth (factset count)

If you want to retrieve a particular set fact w/o erasing it, you can use

`^nth(@1 5)`

where the first argument is like `^first` because you also specify how to interpret the answer and the second is the index you want to retrieve, eg.,

```
^nth(@0object 5)
```

An index out of bounds will fail. Factsets are always numbered 1...n, so the first element is, in fact, `^nth(@0object 1)` would correspond to `@0object` or `^first(@0object)`

Similarly you can do `nth(~concept 2)` to retrieve the third member of a concept (numbering starts at 0).

You can also do `nth` of a JSON object (returns the factid of the nth key/value pair) or JSON array (returns the factid of the nth index/value pair).

`^unpackfactref`

examines facts in a set and generates all fact references from it. That is, it lists all the fields that are themselves facts.

```
@1 = ^unpackfactref( @2)
```

All facts which are field values in `@2` go to `@1`. You can limit this:

```
@1 = ^unpackfactref(@2object)
```

only lists object field facts, etc

`^save ( factset boolean )`

Unlike variables, which by default are saved across inputs, fact sets are by default discarded across inputs. You can force a set to be saved by saying:

```
^save(@9 true) # force set to save thereafter
^save(@9 false) # turn off saving thereafter
```

`^makereal ( {set , factid} )`

If you give this a factset, it will convert any transient facts in that set into permanent. If you give this a factid, it will convert all transient facts created after that id into permanent. This might allow you, for example, to call ‘^jsonopen and get back a transient JSON structure and after inspection you could convert it to permanent if you wanted to.

^setFactOwner ({set , factid} idbits)

In multibot deploys, allows you to change which bot owns this set of facts. Useful if you want to allow a bot to share facts owned by another bot.

^addproperty (set flag)

Add this flag onto all facts in named set. Typically you would be adding private marker flags of yours. If set has a field marker (like `@2subject`) then the property is added to all values of that field of facts of that set.

^conceptlist (kind location)

`^conceptlist(kind location {filter})` generates a list of transient facts for the designated word position in the sentence of the concepts (or topics or both) referenced by that word, based on kind being CONCEPT or TOPIC or BOTH.

Facts are `(~concept ^conceptlist location)` where location is the range location in the sentence (start <<8 + end).

```
^conceptlist( CONCEPT 3) # absolute sentence word index
^conceptlist( TOPIC _3) # whereever _3 is bound
```

Otherwise, if you don't use an assignment, it stores into set 0 and fails if no facts are found. Any set already marked `^Addproperty(~setname NOCONCEPTLIST)` will not be returned from `^conceptlist`.

Special preexisting lists you might use the members of to exclude include: `~pos` (all bits of word properties) `~sys` (all bits of system properties) and `~role` (all role bits from pos-tagging). Only one instance of a concept or topic will be returned as a fact.

If a concept reference covers multiple words (like an unmerged New York City), the concept is indexed at the first word. The location returned is: `(start <<8) | end` .

If you omit the 2nd argument (location), then it generates the set of all such in the sentence, iterating over every one but only doing the first found reference of some kind. If you use `^mark` to mark a position, both the word and all triggered concepts will be reported via `^conceptlist`.

But if the mark is a non-canonical word, mark does not do anything about the canonical form, and so there may be no triggered concepts as well. (Best to use a canonical word as mark).

If you add the optional 3rd argument, it will filter concepts to be only those that start with the filter characters.

```
@0 = ^conceptlist(CONCEPT _0  ^"~bot-")
```

retrieves only concepts that start with `~bot-` .

^purgeboot (what) (see Advanced Layers manual)

`^wordinconcept ( word conceptname )`

Takes any casing of a word and finds which casing in the dictionary is a member of the concept. When you memorize a word, you will get how the user spelled it. Eg., Ebay, when the dictionary actually has eBay. The correct spelling of the word can be found this way as a member of a concept. And composite words using either spaces or underscores can be found as well and returns the correct notation.

`^createattribute ( subject verb object flags )`

This is just like `^createfact`, except that it only allows one fact with this subject and verb to exist.

It will kill off any other such facts. If, for example, you had a fact `(car1 cost $1500)` and executed

`^createattribute(car1 cost $1000)`

then after this the $1500 fact would no longer exist and only the new price fact would exist.

Note if you have facts that reference facts that would be killed off, the createattribute call will decline to create a new fact and fail instead. Also, don't have those old facts as values of variables or factsets because those values will become erroneous. The system will not stop you, but you cannot guarantee the results after that. BE CAREFUL you don't create facts where the verb and object are intended to be constant and the subject varies. It won't work correctly.

`(car space 10)` - fine if 10 can vary

`(10 space car)` - wrong if 10 can vary

See also `^revisefact` which is probably easier to use for most cases.

`^createfact ( subject verb object flags )`

The arguments are a stream, so `flags` is optional. Creates a fact of the listed data if it doesn't exist (unless flags allows duplicates). Or `^createfact($$tmp)` or some other variable that evaluates to a fact stream will also create/find a fact. `$$tmp` might have been written previously using WriteFact.

`^writefact ( F )`

Given a fact index such as might be returned by `first(@1fact)`, writes out the fact in std text notation (such as done by ^export or written into user files). (see `^createfact`).

`^revisefact ( factid subject verb object )`

The existing non-dead user fact will have fields replaced when arguments are not `null`. You cannot change type of field, so a fact subject will require a factid as subject, etc.

`^delete(set)`

erase all facts in this set. This is the same as `^addfactproperty(set FACTDEAD)`.

`^field ( fact fieldname )`

given a reference to a fact, pull out a named field. If the fieldname is in lower case and the field is a fact reference, you get that number. If the fieldname starts uppercase, the system gives you the printout of that fact. Eg for a fact:

`$$f = createfact (I eat (he eats beer))`

`^field( $$f object)` returns a number `(the fact index)` and `^field($$f Object)` returns `(he eats beer)` as the translation of the fact into text.

Fields include: `subject`, `verb`, `object`, `flagsv,`all‘ (spread onto 3 match variables, raw (spread onto 3 match variables).

`all` just displays a human normal dictionary word, so if the value were actually `plants~1` you’d get just plants whereas raw would return what was actually there `plants~1`.

You can also retrieve a field via `$$f.subject` or `$$f.verb` or `$$f.object`.

`^find ( setname itemname )`

given a concept set, find the ordered position of the 2nd argument within it. ^Output that index (0-based). Used, for example, to compare two poker hands.

`^findmarkedfact ( subject verb mark )`

given the arguments, start at subject, follow all facts having the verb, and stop if you can find a fact with the mark given.

`^first ( fact-set-annotated )`

Retrieve the first fact. You must qualify with what you want from it. Retrieve means the fact is removed from the set. `^first(@0subject)` retrieves the subject field of the first fact. Other obvious qualifications are verb, object, fact (return the index of the fact itself), all (spread all 3 fields onto a match variable triple, raw (like all but all displays just a normal human-readable word like plant whereas raw displays what was actually there, which might have been `plant~1`).

`^flushfacts ( factid )`

kills all facts created after this one. To use effectively, you need to create an initial dead fact e.g, `$$marker = ^createfact(junk marker data FACTDEAD)` and then if you want to cancel sentence processing because, for example, you intend to replace this sentence with a new one (like with pronoun resolution), you can erase any facts you created while doing this sentence by doing `^flushfacts($$marker)`.

`^gambittopics ()`

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

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

Otherwise, if you don't use an assignment, it stores into set 0 and fails if no facts are found.

`^intersectfacts ( from to )`

Sees what facts in the from set are in common with the to set. You specify what field to intersect on by naming a field of the to set (or none). Eg.,

```
^intersectfacts(@0 @1object)
```

will find facts in set 0 whose objects match any in set 1. If you don't name a field, you have to find exact matches on the entire fact. You need to assign the result to a new fact set, which will contain all matching facts from the from set.

```
@2 = ^intersectfact(@0 @1object)
```

^keywordtopics ()

Lists topics and priority values for matching keywords in input. An optional argument if `gambit`, will ignore topics without available gambits. The verb used is: `^keywordtopics`.

Note: it does not attempt to match the topic you are currently in, as the normal control scripts should already have tried that topic before coming to the more random thrashing of ^keywordtopics.

^last (fact-set-annotated)

Retrieve the last fact - see `^first` for a more complete explanation.

^length (word)

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

^makereal ()

Convert all user facts that are transient into non-transient facts. Probably only useful when using plans, which generate transient facts representing the state of the world and you want those planned world facts to become the current real facts.

^next (FACT fact-set-annotated)

Allows you to walk a set w/o erasing anything. See `^first` for more complete description of annotation, the distinction between next and `^first` is that next does NOT remove the fact from the set, but moves on to each fact in turn. You can reset a set with

```
^reset(@1)
```

then loop thru it looking at the subject field with

```
loop() { _0 = next(FACT @1subject) }
```

`^pendingtopics ()`

List of currently pendings topics (interesting)

`^pick ( ~concept )`

Retrieve a random member of the concept. Pick is also used with factsets to pick a random fact (analogous to `^first` with its more complete description).

`^querytopics ( word )`

Get topics of which word is a keyword and which are not system topics and which have gambits (not necessarily unused), returns as fact triples of word, "a", topicname. If used in an assignment to a set, it will not fail, but it may return 0 elements. If not used in an assignment, then it will use set @0 and will FAIL if no topics are found.

`^removeproperty ( set flag )`

remove this flag from all facts in named set. Typically you would be removing private marker flags of yours or making transient facts permanent.If set has a field marker (like `@2subject`) then the property is added to all values of that field of facts of that set.

`^reset ( @1 )`

reset a fact set for browsing using ^next.

`^query ( kind subject verb object )`

see writeup earlier.

`^save ( set )`

mark set to be saved with user data from here on

`^sort ( set )`

sort the set.. doc unfinished.

^unduplicate (set)

remove duplicate facts from this set. The destination set will be named in an assignment statement like:

```
@1 = ^unduplicate(@0)
```

Normally this merely removes duplicate facts. If you specify a field as well, no facts having that field duplicated will be kept either. Eg

```
@1 = ^unduplicate(@0object)
```

^uniquefacts (from to)

Sees what facts in the from set are not in common with the to set. You specify what field to intersect on by naming a field of the to set (or none). Eg.,

```
^uniquefacts(@0 @1object)
```

will find facts in set 0 whose objects do not match any in set 1. If you dont name a field, you have to find exact matches on the entire fact not in the 2nd set.

^unpackfactref (set)

Find all facts in set which have facts as fields and then make THOSE facts be the facts of the set. The destination set will be named in an assignment statement like:

```
@1 = ^unpackfactref(@0)
```

^changebot(botname botid)

^changebot(botname botid) allows a bot to pretend to be another bot and access its data, functions, and topics. Variables are not affected by this. The user topic file will remain as the user came into the server.