

## Table of Contents

<b>Topic Functions</b>	<b>2</b>
<b>Marking Functions</b>	<b>5</b>
<b>Input Functions</b>	<b>7</b>
<b>Number Functions</b>	<b>8</b>
<b>Output Functions</b>	<b>9</b>
<b>Control Flow Functions</b>	<b>11</b>
<b>Word Manipulation Functions</b>	<b>13</b>
<b>Multipurpose Functions</b>	<b>17</b>
<b>Facts Functions</b>	<b>19</b>

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.

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

*t: name(xxxx ) This is ambiguous. Is it function call or label and pattern?*

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

<i>t: ^name(xxx )</i>	<i># explicilty say it is a function</i>
<i>t: () name(xxx )</i>	<i># explicitly add an empty pattern</i>

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

**^gambit( value ?)** - If value is a topic name, runs the topic in gambit mode to see if any gambits arise. 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 2<sup>nd</sup> 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.

If the value is omitted entirely, the system will walk the pending topics list most recent first, trying each in turn to find a gambit. As it tries them it uses them up, so ones that fail to produce a gambit will disappear as pending topics for the future.

If the value is an ordinary word, the system will try to find topics with it as a keyword and execute those to get a gambit.

**^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 it can be used or null if it has been erased.

**^hasgambit(topic)** – fails if topic does not have any gambits left unexecuted. Even if it 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
```

**^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. 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()** – see if the prior input ended with a potential rejoinder rule, and if so test it on the current sentence. If we match and don't fail on a rejoinder, the rejoinder is satisfied. If we fail to match on the 1<sup>st</sup> 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.

**^rejoinder(tag/label)** – 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( topic-name ?)** - tests the sentence against the named 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. You can supply

an optional 2<sup>nd</sup> argument FAIL, in which case it will return FAILRULE\_BIT if it didn't fail but it didn't generate any new output either.

If the first argument 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 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.

**^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 2<sup>nd</sup> 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 2<sup>nd</sup> or 3<sup>rd</sup>) 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, perhaps with rejoinders, one way to do that is with this:

s: COMMON (?) some answer

a: () some rejoinder...

You make ^reuses go to COMMON (or whatever you name it) or even ^SETREJOINER 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.

**^setrejoiner({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.

^Setrejoiner does not jump anywhere. It establishes the context for ^rejoiner. 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 ^rejoiner. 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.

**^topicflags(topic)** – given a topic name, return the control bits for that topic. The bits are mapped in dictionary\_system.h as TOPIC\_\*

## Marking Functions

**^mark ( word location)** – 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. 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. With generic you disable or reenale 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.

**^mark (~meat \_0)** – This marks ~meat as though it has been seen at wherever 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.

**^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()`** - restore a global `^unmark(0)` exactly as it was before the global `unmark`. Used typically to mark a topic to be visited when some idiomatic phrase implies it but keywords wouldn't. E.g.,

*?: (what do you do ) ^mark(~occupation)*

**`^unmark` (word match-variable)** – the inverse of specific `^mark`, this takes a match-variable 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.

**`^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(* match-variable)`** says turn off ALL matches on this location temporarily. 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.

**`^mark(* _0)`** to restore all marks to some area.

**`^unmark(*)`** turns off matching on all words of the sentence.

**`^mark(*)`** restores all marks of the sentence.

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()**.

**^marked(word)** - returns “1” if word is marked, returns FAILRULE\_BIT if the given word is not currently marked from the current sentence.

**^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.

**^setposition(value)** – sets the current match position to the numeric position in the sentence given by value. If value was returned as a “both” from **^position**, then start and end are set to it. Otherwise start and end are both set to the simple number given. A “both” value is encoded as the lowest byte is the start and the next higher byte is the end.

**^setposition(\_var start end)** – sets the match location data of a match var to the number values given.

## Input Functions

**^analyze(stream)** – the stream generates output (not printed to user) and then prepares the content as though it were current input. 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.

**^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).

**^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 )** - removes 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?”).

**^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.



## 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

+	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)** – This converts time in seconds (Unix epoch time) from the standard baseline in 1970, 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.

**^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: seconds, minutes, hours, date in month, month name, year, day name of week.

**^timetoeconds(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.

## 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 outputting 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. OUTPUT\_EVALCODE is automatic, so not particularly useful. Useful ones would control how print decides to space things.

**^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.

## 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 out-of-band 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 alternative 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.

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

**^command( args )** – execute this stream of arguments through the : command processor. You can execute debugging commands through here. E.g.,

**^command( args )** – execute this stream of arguments through the : command processor. You can execute debugging commands through here. E.g.,

**^command(:execute ^print(“Hello” )**

**^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:

**CALL** – stops the current outputmacro w/o failing it.

**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.

**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 string enclosing “ from a 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:

**RULE** –stops the current rule and cancels pending output

**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.

**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.

**^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
```

**^nofail(code ...script...)** – the antithesis of fail(). It takes a code and a 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:

**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 propagate outside of nofail.

**INPUT** - no failure propagates outside of the script

**^nonnull(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.

**^addcontext(topic label)** - Sets a topic and context name for use by ^incontext. The label doesn't have to correspond to any real label. The topic can be a topic name or ~ meaning current topic.

**^incontext(label)** – 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. InContext will return the volley index (normal return) if the label has output within the 4 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 4 volley context and are used in normal rule patterns.

## Word Manipulation Functions

**^burst( {wordcount 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\_character is omitted, it is presumed to be BOTH “\_” (which joins composite words and names) and “ ”, which separates words. 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.

*^burst* takes an optional first parameter “wordcount”, which tells it to return how many words 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.

**^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*.

**^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. An optional fourth argument “insensitive” will match insensitively.

**^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.

**^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 1<sup>st</sup> & 3<sup>rd</sup> choices:

conjugate	pos-integer (as returned from ^partofspeech)
raw	integer 1 .. %length (returns the original word in sentence)
syllable	word
hex64	integer-word
hex32	integer-word
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 past_participle infinitive past present3ps present
verb	verb match noun - returns noun form matching verb (sing./plural)
aux	auxverb pronoun - returns verb form matching pronoun supplied for "do", "have", "be"
pronoun	word flip - changes person form for 1 <sup>st</sup> and 2 <sup>nd</sup> person
adjective	word more most
adverb	word more most
noun	word proper lowercaseexist uppercaseexist singular or a number == 1 plural or a number > 1 irregular - 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/uppercase	word
lowercase	word
allupper	word
canonical	word
integer	floatnumber generate integer if float is exact integer

**^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.

**^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.

**^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**. 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.

*^substitute(w “I love lovely flowers” love hate) outputs I hate lovely flowers*

*^substitute(c “I love lovely flowers” love hate) outputs I hate hatefully 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 *I*, 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.

**^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.

**^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.

**^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 PREPOSTION). 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. It is really only useful during the building of the dictionary itself. Value should be all upper case.

**^walkdictionary('function)** calls the named output macro from every word in the dictionary. The function should have 1 argument, the word.

**^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)
    ...
}
```

## 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).

**^enable (what ?)** What can be “topic” or “rule” or “save” or “write @set”. If topic, the next argument can be a topic name or the word “all” for all topics . Designated topics will be enabled (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.

**^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 \$ or \_var, it will be evaluated and then pick will be tried on the result (but it won't recurse to try that again).

**^reset ( what ? )** – what can be *user* or *topic* or *factset*. 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.

## 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:

**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).

**Unipropagate** – 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. To set 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))

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

For unipropagate, if you have these concepts;

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

Then ^query(unipropagate 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)

Unipropagate 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

**^last** ( fact-set ) – retrieve the last fact

**^pick** ( fact-set) – retrieve a random fact

e.g.

`_1 = ^first(@1all)`

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.

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).

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

**^delete**(`@1`) – 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`)

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

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

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

`^nth(@1 count)` 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. An index out of bounds will fail.

**`^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

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*

**`^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)`** 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 location in the sentence.

`^conceptlist( CONCEPT 3)` *# absolute sentence word index*

`^conceptlist( TOPIC _3)` *# wherever \_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 proerties) and `~role` (all role bits from pos-tagging).

If you omit the 2<sup>nd</sup> 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).

**`^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

**^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).

**^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)

Fields include: subject, verb, object, flags, 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*.

**^find( setname itemname)** – given a concept set, find the ordered position of the 2<sup>nd</sup> argument within it. ^Output that index. 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.

**^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`.

**^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 pending 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, returns as fact triples of word, “member”, 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)

**^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.,

^intersectfacts(@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 2<sup>nd</sup> 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)

^