ChatScript Fact Manual

© **Bruce Wilcox, gowilcox@gmail.com**
Revision 10-18-2012 cs5.94

SIMPLE FACTS

Facts are triples of subject, verb, object – though those are just the names of the fields and may or may not be their content. Facts look like this:

> *(Bob eat fish)*

The system has a number of facts it comes bundled with and others can be created and stored either from compiling scripts, or from interactions with the user. Facts can use words, numbers, or other facts as field values, representing anything. You can build records, arbitrary graphs, treat them as arrays of data, etc.

Simple Creating Facts

^createfact(subject verb object) – this creates a fact triple. The system will not create duplicate facts. If you have a fact *(Bob eat fish)* then executing

> *^createfact(Bob eat fish)*

will do nothing further (but it will return the found fact). One way to create a fact of a fact is as follows:

> *^createfact((Bob eat fish) how slowly)*

The other way is to assign the value of fact creation to a variable and then use that variable. You need to pass in a flag at creation, to tell the system the value is a factid.

> *$fact = ^createfact(Bob own fish)*
> *^createfact ($fact Bob pet FACTSUBJECT)*
> *$fact = ^createfact(Bob own dog)*
> *^createfact ($fact Bob pet FACTSUBJECT)*

The above creates facts which are findable by querying for pets Bob has. You can have any number of flags at the end. Other flags include:

> FACTVERB and FACTOBJECT
> FACTTRANSIENT – the fact will disappear at the end of this volley
> FACTDUPLICATE – allow this fact to be a duplicate of an existing fact

Accessing Facts

To find facts, you need to make a query. There can be many different kinds of queries.

^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_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.

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.

System-reserved verbs

The system builds the Word-net hierarchy using the verb *isa*, with the lower-level (more specific) word as subject and the upper-level word as object. E.g.,
        (dog~1 isa animal~4)
The system builds concept and topic sets using the verb *member* with the member value as subject and the set name as object. E.g.,
        (run member ~movementverbs)

When you build a table and a data member has a short-form like *Paris* for *Paris,_France*, the verb is also *member* with subject as short form and long form as object. E.g.,
        (Paris member Paris,_France)

@Fact-Sets

The results of queries are stored in a fact-set. Fact-sets are labeled @0, @1, etc. through @19. By default in the simplest queries, the system will find all facts that match and store them in fact-set @0. A fact set is a collection of facts, but since facts have fields (are like records), it is also valid to say a factset is a collection of subjects, or verbs, or objects. Therefore when you use a factset, you normally have to specify how you want it used.
@1subject means use the subject field
@1verb means use the verb field
@1object means use the object field
@1fact means keep the fact intact (a reference to the fact) – required if assigning to another set.
@1+ means spread the subject,verb,object onto successive match variables – only valid with match variables
@1- means spread the object,verb,subject onto successive match variables– only valid with match variables
@1all means the same as @1+, spread subject,verb,object,flags onto match variables.
 _6 = ^first(@1+)   - this puts subject in _6, verb in _7, object in _8

        ?: (do you have a pet ^query(direct_sv I pet ?)) I have a @0object.
If the chatbot has facts about what pets it has stored like *(I pet dog)* and *(I pet cat)*, then the rule can find them and display one of them. Which one it shows is arbitrary, it will be the first fact found.

You can transfer the contents of one fact-set to another with a simple assignment statement like *@2 = @1* .

You can transfer fields of a fact from a fact-set using assignment, while simultaneously removing that fact from the set. The functions to do this are:
^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)

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) – 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.

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

If you want to know how many facts a fact-set has, you can do this:
^length(@1) - outputs the count of facts

A special count function exists to tell you how many members a concept set has.
^countmember(concept-topic-name) – this will tell you how many members are in this concept or topic at the top level (not recursive).

^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*

Fact Indexing

A fact like *(bird eat worm)* is indexed by the system so that *bird* can find facts with bird as the subject or as the verb or as the object. Similarly *eat* can find facts involving it in each position. As a new fact is added, like *(bird hate cat)* the word *bird* gets the new fact added to the front of its list of facts involving bird in the subject field. So if you search for just one fact where bird is the subject, you get the most recent fact. If you search for all facts with bird as the subject, the facts will be stored in a fact set most recent first (lowest/earliest element of the fact set). You would use ^first(@2) to get its most recent fact and ^last(@2) to get its oldest fact.

Tables

With the ability to create and manipulate facts comes the need to create large numbers of them conveniently. This is the top-level declaration of a table, a combination of a transient output macro declaration and a bunch of data to execute the macro on. Usually the macro creates facts.

The table has a name (ignored- just for your documentation convenience), a list of arguments, a bunch of script, a DATA: separator, and then the table data. The data is line-oriented. Within a line there are no rules about whitespace; you can indent, tab, use lots

of spaces, etc. Each line should have as many elements as the table has arguments. The table ends with the end of file or a new top-level declaration. E.g.,

Table: authors (^author ^work ^copyright)

  ^createfact(^author member ~author)   # add to concept ~author
  ^createfact(^work member ~book)   # add to concept ~book
  ^createfact(^work exemplar ^author)   # author wrote this book
  *if (^copyright != *) { ^createfact(^copyright date ^work) }*

Data:
"Mark Twain" "Huckleberry Finn" 1884
*"Mark Twain" "Tom Sawyer" *   # don't know the date*

For tables with really short data, you can choose to cheat on the separate line concept, and separate your entries with \n , which is the equivalent.

 DATA:
 a 1 \n b 2 \n c 3 \n d 4 \n e 5   # values assigned to letters.
 f 6 g 7

Tables of only single values do not need a line separator. E.g,
table: mine(^arg)
DATA:
value1 value2 value3
value4 value5 value6

A table allows you to automatically list shortened synonyms of proper names. For example, *Paris* could be a shortened synonym for *Paris, France.* In a table of capitals, you would normally make the fact on the full name, and write the shortened synonyms in parens. You may have more than one:

 "Paris, France" (Paris "City of Love") France

These synonyms are represented using the *member* verb, sort of like making a concept set of the full name. The system detects this specially during inferencing, and if an argument to ^query were *Paris*, it could automatically transfer across and consider facts for *Paris,_France* as well. It would not go the other way, however, so if the argument were *Paris_France*, it would not move over to *Paris.* You should store your facts on the full name. The mechanism allows user input to use the short name.

While a line of table data must fill all fields of the table exactly (no more or less), you can tell the system to fill in the remaining arguments with "*" by putting "…" as your last value. Eg.

 table: test(^item1 ^item2 ^item3 ^item4)
 ....
 Data:
 lion 50 ...

This table will use * for item3 and item4 of lion.

Note: If you create member facts to add something to a concept, the concept must have been predeclared. You can declare an empty concept just before the table like this:

concept: ~newconcept()
table: mytable(^x)
createfact(^x member ~newconcept)
DATA:

TABLEMACRO:

When you have tables you generate over and over again, you don't want to repeat all the script for it. Instead you want to declare a permanent table function using a table macro. It looks a lot like a table definition, except it has a different declaration header and has no DATA: or data attached.

> tablemacro: ^secondkeys(^topic ^key)
> $$tmp = join(^topic . 1)
> CreateFact(^key member $$tmp)

The table macro can declares more arguments than the table will have. When you invoke an actual table using it, you will be supplying some of the arguments then, and the rest come from the table data. An invocation of this tablemacro would look like this:

> table: ^secondkeys(~accidents)
> repair garage insurance injure injury

Note several things. This is declared as a table. The system can tell the difference because the table name (^secondkeys) will already have a definition. The arguments you supply must be real arguments, not ^xxx names of dummy arguments). This table presupplies one argument (~accidents). There is no need for a DATA: line because the table function has already been defined- it knows all its code. So one proceeds directly to supplying table data. In this instance, the code will be expecting each table entry is one value, because the ^secondkeys tablemacro said there are two arguments. Since one is presupplied, the table data must supply the rest (1). So this will execute the table code on each of the 5 table data entries.

<center>**Fact Functions**</center>

Various functions create, destroy and aggregate facts, as well as mark and unmark them. For those routines that aggregate facts, the result is stored into a fact set. Usually this is done by assignment, e.g.,

> @2 = gambitTopics()

Such assignments never fail, they just may assign a zero length to the result. Often, however, you can use the function to simultaneously store and test. If not in an assignment context, the function will store results into @0 and fail if the result is no facts. Eg.,

> if (gambitTopics()) { first(@0object)}

^intersect(word1 word2) Sees what words in the first string are in common with the 2nd string. The intersecting words are the subjects of facts stored by assignment. If not done within an assignment context, facts are stored into set 0 and if there is no intersection, this returns a fail code.

^keywordtopics() – gets the most salient topics covering the input (excluding system, or disabled topics). Usually stored into a fact set as facts, if not within an assignment will use set 0 and fail if no facts are found. Facts are most salient first. A fact looks like: (~mytopic topic 22) where higher numbers are more salient. Access via:
>   if (keywordtopics()) { ^gambit(first(@0subject))}

^topics() - returns a list of all topic names as facts. (~topic topiclist ~topic)

^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.

^AddFactProperty(set flag) – add this flag onto all facts in named set. Typically you would be adding private marker flags of yours.

^RemoveFactProperty(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.

^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).

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

^export(filename @setid) - Filename is either a full path including drive or a path below the chatscript root directory. All non-dead, non-null facts in the set will be written to the file, one fact per line. If the file cannot be written or the setid is illegal, it returns FAILRULE. The flags of the fact will NOT include TRANSIENTFACT.

^fact(set field) – given a reference to a fact, pull out a named field.

^find(setname itemname) – given a concept set, find the ordered position of the 2nd 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) – retrieve the first fact

^import(filename @setid erase transient) - Filename has the same interpretation. The second argument names the set where the digested facts will be stored. It is not cleared so incoming facts augment it. And if a fact repeats one already existing, no new fact is created but the set will have the old fact added to it.The third argument, if the word erase, will erase the file after reading it. Any other value is ignored. The fourth argument, if transient, will mark the facts as transient and they will disappear automatically at the end of the volley (so you better process them immediately). Any other value will be ignored and the facts will enter the user's space permanently. If the file cannot be read, it returns FAILRULE.

A rule file contains one rule per line. Rule format is like DICT/facts.txt

(subject verb   object)

or

(subject verb object flags)

The fields subject, verb, and object should contain no embedded blanks, unless the entire value is encased in double quotes. The fields may be normal or they may be facts themselves. E.g.
((_ ~foods My_favorite_cuisine_is_Chinese.) benfavorite cuisine 0x80)

Flags, if supplied, should be a hex number or a normal integer number, representing property bits you want enabled (see values in facts.h).

^intersectfacts (mainstore otherstore) – The system walks the otherstore list of facts, making appropriate marks. It then walks the mainstore to find data appropriately marked and returns the set of matching facts. You can mark either just the word of a field (e.g., @1subject) or all fields and the fact itself (@1all or @1). Then when checking mainstore, it will look to see if just a specific field is marked (@3subject) or if the fact as a whole is marked (if both otherstore and mainstore say to use the entire fact) or if any field of the fact is marked (if otherstore marked a field and checking mainstore says use the whole fact). If otherstore is a word, not a store, then it will be matched against the field in mainstore facts and matching facts returned to the assignment (eg @2 = ^intersectFacts(@1 @3).

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

^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.

^next(set-what) - allows you to walk a set w/o erasing anything. You can reset a set with
^reset(@1)
 then loop thru it looking at the subject field with

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

^pick (~concept) – retrieve a random member of the concept. Pick is also used with factsets to pick a random fact (see FACTS). 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(@1) – reset a fact set for browsing using ^next.

^query(kind subject verb object) – see writeup earlier.

^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.

^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)

^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)

ADVANCED FACTS

Facts of Facts

Suppose you do something like *CreateFact(john eat (wet food peanuts))*. What happens when you retrieve it into a fact set and then do *_1 = ^last(@1)* and get the fact disassembled onto _1, _2, _3, and *4?* What you get for _3 is a reference to a fact, that is, a number. You can decode that by using *^fact(_3 subject) or ^fact(_3 verb) or ^fact(_3 object)* to get *wet* or *food* or *peanuts*. The first argument to ^fact is a fact number.

You get a fact number if you do _3 = CreateFact(….) and can decode _3 the same way. Naturally this function fails if you give it something that cannot be a fact reference.

ESOTERIC FACTS

Compiled Script Table Arguments

You can specify that a table argument string is to be compiled as output script. Normally it's standard word processing like all English phrases. To compile it, you prefix the doublequoted string with the function designator ^. E.g.,
> *DATA:*
> *~books "this is normal" ^"[script a][script b] ^fail(TOPIC)"*

This acts like a typical string. You pass it around, store it as value of variables or as a field of a fact. Like all other strings, it remains itself whenever it is put into the output stream, EXCEPT if you pass it into the *^eval* function. Then it will actual get executed So. To use that argument effectively, you would get it out of the fact you built and store it onto some variable (like _5 or $value) , and then ^eval(_5) or ^eval($value).

FactSet Remaps

Factset names like @1 are not mnemonic. You can "rename" them as follows:
define: @authors @5 -- whenver the system sees @authors, it will use @5
You can do this within a topic (limited to it) or outside a topic (globally thereafter).

Defining your own queries

The query code wanders around facts to find those you want. But since facts can represent anything, you may need to custom tailor the query system, which itself is a mini-programming language. The full query function is takes nine arguments and any arguments at the end you omit default themselves.

All query kinds are defined in LIVEDATA/queries.txt and you can add entries to that (or revise existing ones). The essential things a query needs to be able to do is:
1. Start with existing words or facts
2. Find related words or facts

3. Mark newly found words or facts so you don't trip over them multiple times
4. Mark words or facts that you want to ignore or be treated as a successful find
5. Store found facts

A query specification provides a name for the query and specifies what operations to do with what arguments, in what order.

An essential notion is the "tag". As the system examines facts, it is not going to compare the text strings of words with some goal. That would be inefficient. Instead it looks to see if a word or a fact has a particular "tag" on it. Each word/fact can have a single tag id, drawn from a set of nine. The tags ids are labeled '1' thru '9'.

Another essential notion is the field/value. One refers to fields of facts or values of the incoming arguments, or direct values in the query script. Here are the codes involved:
1. s = refers to the subject argument or the subject field of a fact
2. v = refers to the verb argument or the verb field of a fact
3. o = refers to the object argument or the object field of a fact
4. p = refers to the propogate argument
5. m = refers to the match argument
6. ~set = use the explicitly named concept set
7. 'word = use the explicitly named word
8. @n = use the named fact set

Each query has is composed of four segments. Each segment is separated using a colon. Each segment is a series of actions, which typically involve naming a tag, a field, and then the operation, and possibly special arguments to the operation.

You can separate things in a segment with a period or an underscore, to assist in visual clarity. Those characters are ignored. I always separate actions by underscores. The period I use to mark the end of literal values (~sets and 'words).

EXAMPLE 1 – PARIS as subject

Consider this example: we want to find facts about Paris. The system has these facts:
        (Paris exemplar France)   and (Paris member ~capital)
Our query will be ^query(direct_s Paris ? ?) which request all facts about a subject named Paris (to be stored in the default output factset @0).

Segment one handles marking and/or storing initial values. You always start by naming the tag you want to use, then naming the field/value and the operation. The operations are:
1. t = tag the item
2. q = tag and queue the item
3. < or > scan from the item, tagging things found (more explanation shortly)

The query *direct_s,* which finds facts that have a given subject, is defined as

1sq:s::

This says segment 1 is *1sq* and segment 2 is *s* and segments 3 and 4 have no data. Segment 1 says to start with a tag of '1', use the subject argument and tag and queue it.

Segment two says how to use the queue. The queue is a list of words or facts that will be used to find facts. In our example, having stored the word *Paris* onto the queue, we now get all facts in which Paris participates as the subject (the *s:* segment)

Segment three tells how to disqualify facts that are found (deciding not to return them). There is no code here, so all facts found will be acceptable.

Segment four tells how to take disqualified facts as a source of further navigation around the fact space. There is nothing here either.

Therefore the system returns the two facts with Paris as the subject.

Example 2 – Finding facts up in the hierarchy

Assume you have this fact (23 doyou ~like) and what you actually have is a specific verb *like* which is a member of ~like. You want to find facts using *doyou* and *like* and find facts where *doyou* matches and some set that contains *like* matches. The query for this is *direct_v<o*, which means you have a verb and you have an object but you want the object to match anywhere up in the hierarchy. *<* , which means the start of the sentence in patterns, really means the left side of something. And in the case of facts and concepts, the left side is the more specific (lower in the hierarchy) and the right side is most general (higher in the higherarchy) when the verb is member.