ChatScript Fact Manual

Revision 11/10/2018 cs9.8

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. Flags include:

flag	description
`FACTSUBJECT`	apply to storing fact ids in the subject field
`FACTVERB`	apply to storing fact ids in the verb field.
`FACTOBJECT`	apply to storing fact ids in the object field.
`FACTTRANSIENT`	the fact will disappear at the end of this volley
`FACTDUPLICATE`	allow this fact to be a duplicate of an existing fact.This is particularly important if you go around deleting facts that might be referred to by other facts. Those other facts will also get deleted.So if you want complete isolation from facts that look the same in some subfact but shouldn't be shared, you'll want that subfact declared `FACTDUPLICATE`.
`AUTODELETE`	on a normal fact means that when that fact is destroyed, if it refers to other facts (FACTSUBJECT, FACTVERB, FACTOBJECT) then those referred facts will also get destroyed. `AUTODELETE` on a json fact tells the system that the value stored as the object field is actually a normal fact id value and that fact should be destroyed when the json fact is destroyed.

Accessing Facts

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

`^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 flag	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)

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.

Note: if an argument to a query is a concept name, it will be expanded to all its constituent members unless you put a quote in front of it. You probably want this:

```
@0 = ^query(direct_vo ? member '~conceptname)
```

`^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) )
```

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

Private queries

You can define your own queries beyond the LIVEDATA ones. In a normal topic file as a top level unit you can do:

```
query: name "value"
```

Name is the name to give your query and the query command string is placed within double quotes.

System-reserved verbs

The system builds the Word-net hierarchy using the verb `is`, with the lower-level (more specific) word as subject and the upper-level word as object. E.g.

```
(dog~1 is 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 `@20`.

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.

fields	description
`@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 sub-ject,verb,object onto successive match variables - only valid with match variables

fields	description
`@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(@1all)` - this puts subject in `_6`, verb in `_7`, object in `_8`

It is legal to store `null` into a factset, and it will generally return `null` for all accesses of that fact.

```
?: (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), ^last(fact-set), ^pick(fact-set)

^first - retrieve the first fact, ^last - retrieve the last fact, ^pick - 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 KEEP)
```

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(fact-set)

```
^delete(@1)
```

all facts in @1 will be deleted and the set erased. You can also delete an individual fact whose id is sitting on some variable

```
^delete($$f)
```

If you merely want to empty a factset, you do

```
@0 = null
```

which does not damage any facts. When you do

```
^delete(@0)
```

you delete all facts within that factset AND all facts which use those facts as part of themselves. Deleted facts are never saved at the end of a volley.

Normally you will be blocked from deleting system facts (ones created by a :build and not by execution of your script). But ^delete(@1 boot)‘ will allow you to do that. Deleted system facts cannot have their space reclaimed (other than by system restart).

^length(fact-set)

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

```
^length(@1)
```

outputs the count of facts.

^nth(fact-set count)

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:

```
^enable(write @9) # force set to save thereafter
^disable(write @9 ) # turn off saving thereafter
```

You can store a fact in a fact set easily.

```
$$tmp = createfact(I love you)
@0 = $$tmp
```

or

```
@0 += $$tmp
```

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.

NOTE: Tables are executed (built) at compile time. And their resulting data and variables are saved in the TOPIC folder along with your compiled scripts and loaded when the server starts up. The data is "owned" by the server and not written out to user files. Also, normally you cannot alter table data on the fly from a user script interaction (true of all facts, not just table facts). In the event you are compiling multiple bots at once, the facts of the table are owned by whatever bot is considered to be compiling that chunk of code.

Variable Argument Tables

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.

Alternatively, you can declare the table variable via:

```
table: ^mytable variable ( ^arg1 ^arg2 ^arg3 ^arg4 )
```

which allows you to not supply all arguments and not use ..., but it means you get no error checking if you failed to supply enough arguments.

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:
# data here
```

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.

Datum

You can use a tablemacro within a topic to declare a single table line.

It must be at the top level, like a `t:` or `u:` rule. E.g.

```
topic: ~mytopic []
t: this is a test

datum: ^secondkeys(~accidents) repair
```

Note that unlike tables that are allowed to run to many entries even on the same line sometimes, a datum will only be allowed to run the tablemacro once.

TAB Tables

Normally each data line defines one entry of the table, filling in all columns. But sometimes it's easier to read a vertically oriented table. You can create them like this:

```
Location    Japan   Tokyo
Location    Japan   Yokohama
```

but that's crowded to read and excess typing. Another thing you can do is:

```
Location    Japan   Tokyo
*           *       Yokohama
```

Where * means use the last seen value from prior entries (you write your table to make that interpretation). Still it's tedious to put in the well mannered . *So there is one other thing you can do. You can make a tab table, where tab characters are automatically convered into space-separated* values instead of being ignored white space. table: ^mytable TAB ($_arg1 $_arg2 $_arg3) ...
DATA: Location Japan
Tokyo
Yokohama

```
is one such table. Looks a lot cleaner. Just be certain you are using tabs and not spaces.
You can write self-checking code in your table to confirm users didn't screw up.
The system variable %tableinput shows the actual line seen by the table function at each ent
```

String processing in Tables

It is common for a string to be a table argument. Any functional string `^"xxx"` stores itse
in an appropriate manner.

They are like regular output - they are literally output script.
Formatting is automatic and you get to make them do any kind of executable thing, as
though you were staring at actual output script.

There is no user context when compiling a table.
As a consequence, if you have table code that looks like this:

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

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

Regular strings, by default, remove their quotes and substitute underscores for spaces.
This is good when the intention is as a composite word, but if the string is to be used as
direct output, you may prefer to retain the quotes and spaces.
You can do this by declaring the argument name with `.KEEP_QUOTES`. E.g.

```
table: ^test( ^my1 ^my2.KEEP_QUOTES)
```

It is particularly important to use the quoted form when the contents includes a concept or
topic name that has underscores because the system cannot tell a spacing underscore from
a significant one.

Simple Fact Functions (more in system functions manual)

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)}
```

`^addproperty`(set flag)

Add this flag onto all facts in named set or onto words.
If you just say

    ^addproperty(@9 USER_FLAG3)

then all facts get that flag on them.
The predefined but meaningless to the system fact flags are
`User_flag4`, `User_flag3`, `User_flag2`, `User_flag1`.

If set has a field marker (like `@2subject`) then the property is added to all values of tha
field of facts of that set, that is, a dictionary word.
The flags must come from `dictionarysystem.h` and the set of property flags or system flags.

`^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) # 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 `^co
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

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).

`^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).
See system functions manual for a bit more on how createfact can process data.

`^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)`
```

fieldname include: `subject`, `verb`, `object`, `flags`, `all` (spread onto 3 match variab
`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 han

`^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 itse
`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 di
what was actually there, which might have been plant~1).

`^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.

`^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) }
```

`^pick`(`~concept`)

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

`^reset`(fact-set)
Reset a fact set for browsing using `^next`.

`^query`(kind subject verb object)
See writeup earlier.

`^sort`(set)
sort the set.

`^unduplicate`(set)
Remove duplicate facts from this set. The destination set will be named in an assignment sta

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

Facts vs Variables

How are facts and variables different? Which should you use?

Facts are persistent.

If you don't create them explicitly as transient, they stay with the user forever.

Variables that don't begin with `$$` are also persistent and stay with the user forever.

There are no limits on the number of variables you can have (none that you need
be aware of) and variable names can be up to 999 characters long.

The limits on user facts that can be saved are defined as a parameter when CS is started up
You can create more facts, but it will only save the most recent limit.

Facts are indexed by subject, verb, object, so you can query to find one.

Variables you have to know the name of it (but it can be composed on the fly).
Facts use up more memory, but can be exported to arbitrary files (and imported).

Facts can represent an array of values, with a field as index.
But so can variables with omposited naming.

So mostly it depends on whether you want to find information by
querying. You have implicit associations of facts by the values of the subject, verb, and
object fields. But you could create a variable name of two of the fields of the fact, if the
third field was really the fact's "value".

WARNING

When you get a reference to a fact, like:

```
    $$tmp = ^first(@1fact)
```

that value is a numeric index into fact space. It is ONLY valid during the current volley.
You cannot insure that it will remain valid across volleys.
The valid ways to access facts across volleys are:

1. rerun `^query(...)` to get a set of factsubject
2. get your fact reference into an @factset and have the set marked to save across volleys
3. save the fact as text to a permanet variable, e.g.,

```
        $fact = ^WriteFact(^first(@1fact)
```

You can later reaccess (or recreate) this fact via `^createfact($fact)`.

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 `^field( _3 subject)` or `^field(_3 verb)` or `^fact(_3 object)`
to get wet or food or peanuts. The first argument to `^field` 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.

Flags

Facts may have flags on them. You can create them with flags (see `^createfact`) and you
can get them using ^field or when you spread out a fact onto a collection of match variables

System-defined flags (which should not be set or erased by user scripts) are:

flag	description

`FACTSUBJECT`
`FACTVERB`
`FACTOBJECT`
	describe that a field is a fact

`FACTDEAD`	indicates the fact has been killed and will go away at the end of volley

Flags you can use to define facts that have system meaning but may or may not stay on them a

flag	description
`FACTDUPLICATE`	allow multiple versions of the same fact
`FACTTRANSIENT`	the fact should die at the end of the volley if not in a fact set

```
|`ORIGINAL_ONLY`| a "member" fact defining a concept only uses the raw word

Flags you can set for yourself include:

| flag          | description
|---------------|---------------------------------
|`USER_FLAG1`<br>`USER_FLAG2`<br>`USER_FLAG3`<br>`USER_FLAG4` | user defined

Facts created by JSON code have user markings also, renamed as

| flag          | description
|---------------|---------------------------------
|`JSON_PRIMITIVE_VALUE`<br>`JSON_STRING_VALUE`<br>`JSON_ARRAY_VALUE`<br>`JSON_OBJECT_VALUE`
|`JSON_ARRAY_FACT`<br>`JSON_OBJECT_FACT`| which indicate what kind of value the subect of th


## JSON

Json is a great representation for creating and accessing complex fact structures.
See the [ChatScript JSON manual](ChatScript-Json.md) for more details.


# 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 outpu
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:

    rename: @bettername @12
```

in a script before any uses of `@bettername`, which now mean `@12`. Then you can do:

```
    $$tmp = @betternamesubject
```

Predefined queries

`exact_` - use the given arguments without expanding beyond them. Most efficient when you know exactly what you want.

`exact_svrange` - find facts given subject and verb, where object is a number and must be in
`exact_vrange` find facts given verb, where object must be number in range (>= propogate <=

e.g. @0 = ^query(exact_vrange ? myverb ? -1 ? ? $_lowvalue $_highvalue)

`direct_` and `direct_flag`

`up2set` - propogate upwards from subject to find the fact that leads to object of which is

given: (soap member ~hygiene_items) and (~hygiene_item member ~personal_items) and (~personal_items member ~drugstore) and (~drugstore member ~storetypes) given ^query(up2set soap ? '~store_type 10) - ie, what kind of store sells soap yields (~drugstore member ~storetypes) "'

`up2seta` - similar to up2set but you name the verb instead of assuming member/is

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:

code	meaning
`s`	refers to the subject argument or the subject field of a fact
`v`	refers to the verb argument or the verb field of a fact
`o`	refers to the object argument or the object field of a fact
`p`	refers to the propagate argument
`m`	refers to the match argument
`~set`	use the explicitly named concept set
`'word`	use the explicitly named word
`@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:

operation	meaning
`t`	tag the item
`q`	tag and queue the item
`<>`	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.