ChatScript Fact Manual

© **Bruce Wilcox, gowilcox@gmail.com** brilligunderstanding.com

Revision 4/21/2015 cs5.2**2**

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

– 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.

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:

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.

^query(kind subject verb object count fromset toset propogate 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 doublequotes.

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)

Who owns/sees factsubject

Facts come with the dictionary, can be built during :build0 and :build1, or can be created on the fly during execution of the chatbot. All but on the fly are resident all the time in the engine and visible to all users and all bots. Facts created on the fly are private to a specific user talking to a specific bot. If you need to share on-the-fly facts across users and/or bots, you can write code the exports facts to a file and imports them back in again for different users and/or bots.

@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.
@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(@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 rename a factset (make your own mnemonic label) by using a rename: before any reference.
rename: @betternname @12
in a script before any uses of @betternname, which now mean @12. Then you can do:
$$tmp = @betternamesubject

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)
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 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.

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:

^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.

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:
```

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.

String processing in Tables

It is common for a string to be a table argument. Any functional string ^"xxx" stores itself 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.

Fact Functions

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. 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)     # whereever _3 is bound
```

Otherwise, if you don't use an assignment, it stores into set 0 and fails if no facts are found. Any set already marked ^Addproperty(~setname NOCONCEPTLIST) will not be returned from ^conceptlist. Special preexisting lists you might use the members of to exclude include: ~pos (all bits of word properties) ~sys (all bits of system proerties) and ~role (all role bits from pos-tagging).

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).

^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

^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)

^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 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-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 dont 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
ˆreset(@1)
then loop thru it looking at the subject field with
loop() { _0 = next(FACT @1subject) }

^pendingtopics() - list of currently pendings topics (interesting)

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

^queryTopics(word) – get topics of which word is a keyword, 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.

^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 2nd set.

^unpackfactref(set) - find all facts in set which have facts as fields and then make THOSE facts be the facts of the set. The destination set will be named in an assignment statement like:

@1 = ^unpackfactref(@0)

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 (default 100). 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 composited 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”.

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.

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.