ChatScript JSON Manual

Real World JSON

JSON (JavaScript Object Notation) is an open standard format using human-readable text to transmit data objects over the web. It is a common standard largely replacing XML which is too wordy and hard to read. JSON has primitive datatypes (text without doublequotes that cannot contain any whitespace). These are null, false, true, integers, and floats. The string datatype is a sequence of characters in double quotes. JSON also has two datatypes that represent collections of values, the array and the object. A JSON array is a list of JSON datatypes separated by commas and placed within square brackets [], e.g.,

```
[ A, 2, [ help, life], [] ]
```

Indices of an array start at 0, so the above has as values:

```
[0] = A
[1] = 2
[2] = an array of 2 values
[3] = an empty array
```

Note that arrays can hold values of different types. Arrays have no hole in indexing and usually ordered and retain that order, but they work out of order as well (^Jsonmerge can alter the order).

If you have an array, you can use dot notation to find the index of some value in it, eg

    $_index = $array.space

will find what array index the value `space` is at;

15. given an jsonarray treated as an object (using dot) $array.value returns the index of the element that has that value

A JSON object is a list of key-value pairs separated by commas and placed within curly braces {}, e.g.,

```
{
    "key1": 1,
    "bob": "help",
    "1": 7,
    "array": [1, 2, 3],
    "object12": {}
}
```

Each key is a string (must be encased in double quotes) and joined to an ending colon. Again types can be mixed in the values. `{}` is the empty object, just as `[]` is the empty array. Key-value pairs have no guaranteed order and may shuffle around if you manipulate the structure. You can nest arrays and objects inside each other.

ChatScript & JSON

JSON is an excellent language to represent more complex ChatScript facts as well as interact with the web. ChatScript can convert back and forth between the JSON text string passed over the web and ChatScript facts that represent the structure internally. If you tried to create facts using ^createfact(), you would find making the data shown below extremely difficult and non-obvious. Though it can be done. But as JSON, it is easy to create facts to represent the structure and to access pieces of it.

```
{
    "firstName": "John",
    "lastName": "Smith",
    "age": 25,
    "address": {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "state": "NY",
        "postalCode": "10021-3100"
    },
    "phoneNumbers": [
        {
        "type": "home",
        "number": "212 555-1234"
        },
        {
        "type": "office",
        "number": "646 555-4567"
        }
    ],
    "children": [],
    "spouse": null
}
```

JSON facts are triples where the subject is always the name of a JSON object or array. For objects, the verb of the fact is the name of a field and the object is its value. Such a structure might be composed of these facts: `internal CS (jo-3 name Robert) (jo-3 age 10) (jo-3 gender male)` Note that CS does not need to put double quotes around the values or even the field names (unless the field name has spaces or other unusual characters within it).

This would correspond to a JSON string that looks like this:

```
{ "name": "Robert", "age": 10, "gender": "male"}
```

Actually, the "age 10" fact in ChatScript could represent either what was shown above or:

```
{ "name": "Robert", "age": "10", "gender": "male"}
```

For arrays, the verb of the fact is the numeric index in the array and the object is the value. Such a structure might be composed of these facts:

```
(ja-5 0 Robert)
(ja-5 1 Terry)
(ja-5 2 Paula)
```

This would correspond to a JSON string that looks like this:

```
[ "Robert", "Terry", "Paula"]
```

CS has rich support for creating JSON data. You must create the top level array or object, but after that you can just do assignments somewhat arbitrarily, with CS automatically creating new subobjects or subarrays as needed. E.g. Also note that CS assignments onto object fields do not require you double quote the field name (unless the name as spaces within it or other unusual characters).

```
$_x = ^jsoncreate(TRANSIENT object)
$_x.field[] = 5
$_x.name.age.birthdate = "January 5, 2020"
```

In the above, the scripter has explicitly created a transient JSON object. The 2nd line assignment automatically creates a transient array to save as the value of `field` and then inserts the value 5 into it. The 3rd line assignment automatically creates a field called name with a JSON object, and that object gets a field called age, and that object gets a field called birthdate which is assigned the text string.

JSON CS ANOMOLIES

Because JSON has distinct data types as values (primitives, text strings, etc) and CS represents everything as text strings, there are some issues in moving back and forth in representation. Whereas JSON shows all strings encased in double quotes, when stored as values in CS the double quotes are always removed. And JSON primitives `false`, `true`, and numbers are also just strings. Therefore JSON primitive `2` and the JSON string `"2"` are the same to CS. To maintain data type information, JSON facts have flag bits that describe how to treat the object value of the fact. This is automatically handled by CS and normally you have no reason to pay any attention to it.

But when you copy a value out of the fact onto a CS variable, the type data goes away. This is not a problem for CS, except when you want to store the

CS variable back into a JSON structure. Then YOU become responsible for knowing the type. CS will assume any of its values that match a JSON primitive will be that, so you need to encase them in double-quotes if you want them to be stored as JSON strings. CS will automatically store everything else as JSON strings and when exporting them to text, will automatically encase them in double quotes. For example,

```
$_x = ^jsoncreate(TRANSIENT object)
$_x.value = true          # JSON boolean primitive
^jwrite($_x)              => {"value": true}
$_y = $_x.value
$_x.value = $_y
^jwrite($_x)              => {"value": true}


$_x.value = "true"         # JSON string true
^jwrite($_x)              => {"value": "true"}
$_y = $_x.value
$_x.value = $_y
^jwrite($_x)              => {"value": true}


$_x.value = "true"         # JSON string true
^jwrite($_x)              => {"value": "true"}
$_y = ^join(\" $_x.value \")
$_x.value = $_y
^jwrite($_x)              => {"value": "true"}
```

All this breaks down a bit for the JSON **null** primitive and the empty string ““. CS does not distinguish between having no value and having a value which has no characters. And CS uses **null** to mean clear a value entirely. So to create a JSON null value, use the special word **json_null**. You can use that directly in a JSON assignment, or you can put it as the value of a CS variable which eventually gets used in a JSON assignment.

```
$_x.value = null          # remove the JSON field called value
$_x.value = "null"        # assign text string null to field value
$_x = null                  # remove any value from variable
$_x.value = json_null              # special value meaning json null primitive
```

This just leaves us with the awkwardness of the empty string. You can't store an empty string onto a CS variable. Instead you store a string with two double quotes. Which CS itself always treats as just that, a two-character string. But when assigned into a JSON structure, any double quoted string has its double quotes stripped. So in JSON we CAN have an empty string. No problem yet. The problem comes in the other direction. Assigning this JSON empty string onto a CS variable clears it (it acts like null). And then putting that variable back onto a JSON structure will save JSON null. So you have to manually track what you want, if you want it.

```
$_x.value = ""              # JSON empty string
$_y = 5
$_y = $_x.value        # clears $_y (acts like null)
$_x.value = $_y     # assigns JSON null
```

Note, also, that while JSON is case sensitive in its field names, CS will lookup insensitive. It doesn't matter if you mis-capitalize the name of a field. CS doesn't believe you should have two fields with the same name but different capitalization and will treat them as the same.

JSON is case sensitive in field names. CS is not. If you offer a new case field name and it already has a field with a different case, it will use the preexisting one.

JSON FACTS

Since JSON data structures are implemented as facts, they can use the ˆquery abilities just as ordinary facts can. Whereas in JSON you can only access data going top down, with CS you can equally access data at any level directly querying on some field name. So depending on how you name a field and whether or not the field only exists for a single object, you might be able to query directly to it.

Note that JSON has no mechanism for sharing JSON subtrees. Hence anytime you create a JSON fact structure in CS, the facts will all be unique. You do this like this:

```
    $_tmp = ˆcreatefact($$jsonobject newfield $$jsondata (FACTTRANSIENT JSON_OBJECT_FACT JSO
```

If $$jsonobject is a json object name, you add a field called newfield whose value is a JSON object, thus joining the trees. If you serialize (write into text) these two JSON objects they will become completely separate structures when read back in (JSON doesn't share).

Accessing the web with JSON

You will learn how to create JSON structures below. The other key to a web-enabled CS is ˆjsonopen(). To see how easy it is to talk to the web, just look at this code.

```
# The cleanest way to build json data
$_var = ˆjsoncreate(transient object)
$_var.fieldname = $myvar

An alternative limited to max output buffer size (~80K):
$_var = ˆjsonparse(ˆ"{ fieldname: $myvar }")

And now to send data upstream:
$_url = ˆ"http://myHost:myIP"
```

```
$_userAgent = ^"User-Agent: %bot, ChatScript"
$_header = ^" ~Accept: application/json
                 $_userAgent
                 ~Content-Type: application/json "
$_response = ^jsonopen(transient POST $_url $_var $_header $_userAgent)
```

JSONFLAGS - Optional 1st arg to some JSON routines

Routines that will create facts for JSON will by default create them as transients (they die at end of volley unless you work to save them). You can override this default by saying `permanent` or `transient` or `boot`. This applies to `^jsonopen`, `^jsonparse`, `^jsoncreate`, `^jsonobjectinsert`, `^jsonarrayinsert`, `^jsoncopy`, `jsonmerge`.

Json arrays all start with the name 'ja-' and json objects all start with the name 'jo-'. The next letter indicates the storage media. 't' means transient and the structure dies at the end of the volley. 'b' means boot and the data moves to the boot later at the end of the volley. Anything else (e.g. an immediate digit) means the structure is permanent and is stored in the user's topic file.

You can also add a flag `safe` (described later) to `^jsonparse`, `^jsonobjectinsert`, `^jsonarraydelete`. You can also add a flag `unique` to `^jsonarrayinsert`. You can also add `duplicate` to ^jsonobjectinsert.

When multiple flags are desired, put them into a simple string, `"DUPLICATE PERMANENT"`. Case doesn't matter.

When you want to add a reference to a normal factid (as is returned by ^createfact) you can add the flag `AUTODELETE`

You can also assign user flags by listing `USER_FLAG1`, through `USER_FLAG4` as a flag as well. The JSON fact will have that flag on it, which you can use in conjunction with `^query` to limit what matches can be found.

^jsonstorage(argument)

This expects a json array or object name and returns `permanent`, `transient`, `boot` or fails completely depending on what is given it.

^jsonparse({JSONFLAGS} string {~ignoreconcept { ~underscorecon-cept} })

`string` is a JSON text string (as might be returned from a website) and this parses into facts. It returns the name of the root node JSON composite. This name will look like this:

- `ja-0` – a json array numbered `0`.
- `jo-0` – a json object numbered `0`.
```

Note that the above names are true only for PERMANENT JSON objects. Transient ones will be named like `ja-t0` and `jo-t0`.

As new JSON composites are created during a volley, the numbers increase to keep them all distinct. JSON composites are all created as transient facts and will die at the end of the volley unless you do something to explictly keep them (typically `^jsongather` into some factset and then saving that OR using that to remove all their transient flags OR using `^delete()` to destroy the facts in the set).

If you are keeping JSON across volleys, you should use the optional Json flags argument to make sure numbering never collides (normally the numbers start back at 0 for each new volley).

Real world JSON has stricter requirements on its format than CS does.

While CS will output strict JSON, you can input slack JSON. You do not need to put commas between elements of an array or object. And you do not need to put quotes around a key name. So the following is legal text form:

```
[a b {bob: 1 sue: 2}]
```

Formally JSON has specific primitives named true, false, and null, that systems may care about, but to CS they are nothing special. Numbers in JSON can be integer or have a decimal point and an exponent. CS does not currently support exponent notation, so you can only do: `1325` or `566.23`.

JSON originally required a JSON string be a composite of some kind. Now it allows any JSON type to be a JSON document. CS sticks to the original requirement, because passing around a single value as JSON is pretty useless.

Once you have a JSON fact structure, you can go the opposite direction and convert it back into a string or display it visually.

CS accepts extended JSON syntax for parsing into a json fact structure. Any place you have an object value, you can refer to a ChatScript user or match variable and that value will be substituted in. E.g.,

```
^jsonparse("{ a: $var, b: _0 }")
```

Note you should use a regular quoted string and not a function string like `^"{ a: $var, b: _0 }"`. If you use a function string, substitution will happen before calling ^jsonparse. Which might be a problem if you had something like this:

```
^jsonparse("{ a: $var, b: _0aba }").
```

where you wanted the value of `b` to be `"_0aba"`. Had you used an active string, the _0 would have been replaced with its contents.

Also, you can use json dereference operators to take apart an existing json structure and use values of it in the current one. If $_y points to a json structure, then

```
^jsonparse(" opensquiggle a: $var, b: $_y.e[2] closesquiggle ")
(saying squiggles because Latex print complains if I use real ones)
```

would find a json object reference on `$_y`, get the e field, and get the 3rd array value found there.

An initial argument of `safe` will locate the correct end of the data for json parsing, allowing you to pass excess data. This is important for passing in json data in OOB information. OOB information comes in `[]` and Json itself uses `[]` at times, so it would be impossible to find the correct end of pattern with normal pattern matching. Instead just do:

```
u: ( \[ _* ) and call ^jsonparse(safe _0)
```

This will figure out where the OOB marker actually is and only parse within it. You can add NOFAIL before the string argument, to tell it to return null but not fail if a dereference path fails cannot be found.

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

You can pass JSON data on input via OOB notation:

```
[ {"token": "myid", "accounts": ["whatever"]}]  User message
```

and then have a pattern to grab the OOB data and call `jsonparse` with it (using SAFE as a parameter). OOB input is not subject to human tokenization behavior, spellchecking, etc.

Note: There is a limit to how much JSON you can pass as OOB data nominally, because it is considered a single token. You can bypass this limit by asking the tokenizer to directly process OOB data, returning the JSON structure name instead of all the content. Just enable `JSON_DIRECT_FROM_OOB` on the `$cs_token` value and if it finds OOB data that is entirely JSON, it will parse it and return something like `jo-t1` or `ja-t1` in its place. Eg. `[ { "key": "value} ]` will return tokenized as `[jo-t1]`.

```
u: GIVEN_JSON(< \[  _* \] ^JsonKind('_0)) $$json = '_0
```

then you can do $$json.field directly.

Note: `^jsonparse` autoconverts backslash-unnnn into corresponding the utf8 characters.

Note: You cannot normally send JSON text as normal input from a user because CS tokenization is set to handle human input. So doing something like this as input will not work:

```
doLogin {"token": "myid", "accounts": ["whatever"]}
```

But you can pass JSON data directly as part of user input with warning. But user input normally is limited to 254 tokens AFTER oob is seen. To be able to pass complex JSON data as part of the user message (because the oob is often controlled by host system so not available per user message), you can now put in

user input the word `json` and follow that with a JSON structure of arbitrary size. That structure will be completely converted to the corresponding JSON internal fact structures and the simple JSON structure name returned (regardless of the setting of tokencontrol). So

```
my input value is JSON {x:y,b:z} here
```

will become

```
my input value is JSON jo-3 here
```

and jo-3 will have the json structure.

If you have given an ignoreconcept, then whenver a field name is encountered in the incoming data matches the name of a member of that concept set, that field and all data below it are discarded and not returned.

If you have given an underscore concept, then any field name encountered in the incoming data that matches a member of that concept, that field's string value will have any spaces in it replaced with underscores.'

jsonformat(string)

Because technically JSON requires you put quotes around field names (though various places ignore that requirement) and because CS doesn't, the function takes in a slack json text string and outputs a strict one.

Accessing JSON structures

Direct access via JSON variables `$myvar.field` and `$myvar[]`

If a variable holds a JSON object value, you can directly set and get from fields of that object using dotted notation. This can be a fixed static fieldname you give or a user variable value or a match variable value (quoted or unquoted): `$myvar.$myfield` is legal, as is `$myvar._0` or `$myvar.'_0`.

Dotted notation is cleaner and faster than `^jsonpath` and `jsonobjectinsert` and for get, has the advantage that it never fails, it only returns null if it can't find the field. If the path does not contain a json object at a level below the top, one will automatically be created if trying to do assignment, and have the same transient/permanent property as the immediately containing object. If the top level variable is not currently an object, assignment will fail. CS will not create an object for you because it doesn't know if it should be transient or permanent.

```
$x = $$obj.name.value.data.side
$$obj.name.value.data.side = 7
```

Similarly you can access JSON arrays using array notation:

```
$x = $$array[5]
$x = $$array[$_tmp]
```

```
$$obj.name[4] += 3
$$obj.$_kind[4] += 3
$$obj._0[4] += 3
$$obj.'_0[4] += 3

$x.foo[] = Bruce
```

If foo is currently undefined, the system will create a JSON array for you, with permanency that matches the JSON object of $x. You cannot do $x[] and have this happen because at the top level the system does not know what permanency to use. Once there is a JSON array in $x.foo, assignments with `foo[]` will add elements to the array. You can designate the index only if it will be the next index in succession.

```
$_x[] = 0     # creates array element 0
$_x[1] = 1  # accepts array element 1 creation request
$_x[3] = 3  # will fail - must be in sequence
```

You can also do

```
$_array1 +=  $_array2 # copies contents of array2 into array1
$_array1 += value     # adds value into array1
```

Adding an element to an array using these notations will automatically select DUPLICATE entries being legal. If you want the insertion to be unique, use ^jsonarrayinsert(UNIQUE xxx xxx) OR in your botmacro change the default by doing

```
$cs_json_array_defaults = #JSON_ARRAY_UNIQUE
```

HOWEVER, if you have explicitly created an array using ^jsoncreate, it will add elements based on how you declared it (UNIQUE or DUPLICATE)

The only restriction on arrays is that you cannot add a new array index value without using ^jsonarrayinsert as you are not allowed to create discontiguous indices.

NOTE If a variable has as its value the name of another variable, CS will attempt to autoindirect through it to get to the JSON structure. if $x ='y' and $y points to a json structure, then $x is indirected and $x.val means $y.val .

NOTE JSON is normally a non-recursive structure with no shared pointers. But ChatScript allows you to store references to JSON structures in multiple places of other JSON structures. This has its hazards. It presents no problem when transcribing to text for a website using `^jsonwrite` . And when you have something like this:

```
$x = ^jsoncreate(object)
$y = ^jsoncreate(object)
$x.field = $y
$x.field1 = $y
$x.field = null
```

Assuming that a JSON structure is not available in multiple places, the assignment of null (or any other value) to a field that already has a JSON structure will normally cause the old value structure to be fully deleted, since it's only reference is removed. And the system does check and delete the structure if it is not referred to by some other JSON field. But there are limits. The system has no idea if you have a pointer to it in a variable. Or if it is part of a pathological indirection sequence like this:

```
$x = ^jsoncreate(object)
$y = ^jsoncreate(object)
$x.field = $y
$y.field = $x
$x.field = null
```

The two structures point to each other, each only once. So assigning null will kill off both structures.

Assigning `null` will remove a JSON key entirely. Having a null value and not having the key are the same from ChatScript's perspective (but not from JSON's)'

Accessing opposite ends of JSON structures   For an array, obviously $array[0] returns one end of the array, and you can get the index of The other end by doing

```
$_tmp = ^length($array) - 1
$array[$_tmp]
```

But a convenient shortcut is

```
$array[-1]
```

For objects, $object[0] returns the name of the most recent field added and $object[-1] returns the name of the first field added.

using a $word as a field name rather than as indirection   Just escape the $.

```
$data.\$varname = hello
```

^jsonkind(item)

If item is the name of a json array, it returns `array`. If item is the name of a json object, it returns `object`. Otherwise it fails.

^jsonmerge({kind} struct1 struct2)

Takes two json arrays or two json objects and adds unique elements of struct2 into struct1 recursively.

11

If `kind` is `key` and the two structs are arrays of objects, then unique objects from the second array are added to the first array (this is the default). Unique meaning that the first property from the object does not exist. If `kind` is `key-value` is like `key` but the uniqueness test is extended to the property value as well. If `kind` is `sum` and the two structs are objects, then the values for the same property name in the two objects are added together. If `kind` is `sumif` is like `sum` but items in `struct1` are only kept if they are also in `struct2`.

`^jsoncount(struct depth)`

Walks the given JSON structure, displaying it, except that for arrays arising after depth, just the count of elements is shown

`^jsonpath( string id )`

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

Array values are accessed using typical array notation like `[3]` and object fields using dotted notation. A simple path access might look like this: `[1].id` which means take the root object passed as id, e.g., `ja-1`, get the 2nd index value (arrays are 0-based in JSON).

That value is expected to be an object, so return the value corresponding to the id field of that object. In more complex situations, the value of id might itself be an object or an array, which you could continue indexing like `[1].id.firstname`.

You can walk an array by using `[$_index]` and varying the value of `$_index`.

When you access an array element, you have to quote the text because it consists of multiple tokens to CS which breaks off `[` and `]`. If you are just accesing an object field of something, you can quote the string or just type it directly

```
^jsonpath(.id $object2)
^jsonpath(".id" $object2)
```

Of course you don't always have to start at the root. If you know what you want is going to become object 7 eventually, you could directly say .id given `jo-7` and the system would locate that and get the id field. Likewise if you know that key names are somehow unique, you could query for them directly using

```
^query(direct_v ? verbkey ?)
```

Or even if the key is not unique you can restrict matches to facts having the `JSON_OBJECT_FACT` flag.

```
^query(directflag_v ? verbkey ? 1 ? ? ? JSON_OBJECT_FACT)
```

Be aware that when `^jsonpath` returns the value of an object key, when the value is a simple word, it just returns the word without doublequotes (since CS

just stores information as a single word). But if the value contains whitespace, or JSON special characters, that may mess up if you pass it to `^JSONFormat`. You can get `^jsonpath` to return dangerous data as a string with double quotes around it if you add a 3rd argument "safe" to the call.

```
^jsonpath(".name" $_jsonobject safe)
```

`^jsonpath`

Generally speaking the direct access methods are clearer than using this function.

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

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

Correspondingly, if you are trying to dump all keys and values of a JSON object, you could do a query like this:

```
@0 = ^query(direct_s $_jsonobject ? ?)
^loop()
{
    _0 = ^first(@0all)
    and then you have _1 and _2 as key and value
}
```

If you need to handle the full range of legal keys in json, you can use text string notation like this `^jsonpath(."st. helen".data $tmp)`.

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

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

`^jsonloop($jsonstruct $var1 $var2)`

This is like ^loop, but explicitly for json data. It is faster and cleaner than writing a loop. A normal loop might look like:

```
    $_count = ^length($_jsonarray)
    ^loop($_count)
    {
        $_count -= 1
        $_value =  $_jsonarray[$_count]
        ...
    }
```

And similarly for a JSON object one might want:

```
$_count = ^length($_jsonobject)
^loop($_count)
{
    $_count -= 1
    $_jsonfield = $_jsonobject[$_count]
    $_value =  $_jsonobject.$_jsonfield
    ...
}
```

But for large numbers of elements (eg a thousand +) this is ineffient because it
has to search for each element each time. The efficient way is:

```
^jsonloop($_jsonobject $_field $_value)
{
    ...
}
```

where the loop runs most recent entry to least recent, putting the field name in
`$_field` and its value in `$_value`. Similarly for a jsonarray it does the index as
`$_field` and `$_value` is its value.

You can choose (or annotate) which way the loop runs by adding an optional
3rd argument OLD or NEW.

```
^jsonloop($_jsonarray $_index $_value OLD) will walk oldest first (i.e. index 0 and increas
^jsonloop($_jsonarray $_index $_value NEW) will walk newest first (i.e. index max and decrea
```

^length(jsonid)

Returns the number of top-level members in a json array or object.

Printing JSON structures

^jsonwrite({plain} name)

name is the name from a json fact set (either by ^jsonpart, ^jsonopen, or some
query into such structures). The result is the corresponding JSON string (as a
website might emit), without any linefeeds. plain is an optional first argument
that makes it write out less noisy JSON that CS already accepts, ie field names
and string values dont need double quotes around them unless they have certain
characters in them (like spaces and semi-colons and escape characters). This
makes data output from CS and intended to be read back into CS easier to read.
So

```
jo-5 which is {"value":"color"}  prints as  {value:color}
```

^jsontree(name {depth} {array})

name is the value returned by ^jsonparse or ^jsonopen or some query into
such structures. It displays a tree of elements, one per line, where depth is

represented as more deeply indented. Objects are marked with `{}` as they are in JSON. Arrays are marked with `[]`.

The internal name of the composite is shown immediately next to its opening punctuation. Optional depth number restricts how deep it displays. 0 (default) means all. 1 means just top level.

Optional "array" means show number of array elements rather than the elements themselves at any depth below depth.

JSON structure manipulation

You can build up a JSON structure without using `^jsonparse` if you want to build it piece by piece. And you can edit existing structures.

`^jsoncreate( {JSONFLAGS} type )`

Type is either `array` or `object` and a json composite with no content is created and its name returned. See `^jsonarrayinsert`, `^jsonobjectinsert`, and `^jsondelete` for how to manipulate it. See writeup earlier about optional json flags.

The system makes ids with different names for build0, build1, and user. And increments ids as it creates new ones. If the id created already exists, the system increments and tries again. If there are lots of collisions you can speed things up by setting $cs_jid. This changes the number to start with.

`^jsonarrayinsert( {JSONFLAGS} arrayname value )`

Given the name of a json array and a value, it adds the value to the end of the array. See writeup earlier about optional json flags. If you use the flag unique then if value already exists in the array, no duplicate will be added.

`^jsonarraydelete( [INDEX, VALUE] arrayname value )`

This deletes a single entry from a JSON array. It does not damage the thing deleted, just its member in the array.

- If the first argument is `INDEX`, then value is a number which is the array index (0 ... n-1).
- If the first argument is `VALUE`, then value is the value to find and remove as the object of the json fact.

You can delete every matching `VALUE` entry by adding the optional argument `ALL`. Like: `^jsonarraydelete("VALUE ALL" $array 4)`

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

If the key has an existing value then if the value is a json object it will be recursively deleted provided its data is not referenced by some other fact (not by any variables). You can suppress this with the `SAFE` flag. `^jsonarraydelete(SAFE $obj $key)`.

`^jsonreusekill( $jsonstruct )`

Kills the facts of this struct and makes them immediately available for reuse as free facts.

`^jsonarraysize( name )`

deprecated in favor of ˆlength

`^jsontext(factid)`

ˆjsontext adds quotes if object of fact is a json text and FAILS if object is not a json fact.

CS represents json text as ordinary unquoted text, but text that looks like float numbers would be subject to possible float truncations or expansion of digits. This preserves it as text.

`^jsoncopy( name )`

Given the name of a json structure, makes a duplicate of it. If it is not the name of a json structure, it merely returns what you pass it.

`^jsonmerge( {creation_arg} control arg1 arg2 )`

ˆjsonmerge take two json structures and perform a top level merge. The result is a copy of the first argument, with top level fields augmented with fields from arg2 not found in arg1. Optional first argument is the standard one for many Json creation functions.

Control describes how to merge fields of json objects. `key` means if arg1 already has this key, ignore the corresponding field from arg2. `key-value` means add in the field and value unless they are the same.

You can also request the system to sum values of fields of objects with control values of `sum` or `sumif`.

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

inserts the key value pair into the object named. The key does not require quoting. Inserting a json string as value requires a quoted string. Duplicate keys are ignored unless the optional 1st argument `DUPLICATE` is given. See writeup earlier about optional json flags.

If the key has an existing value and `DUPLICATE` is not a factor, then if the value is a json object it will be recursively deleted provided its data is not referenced by some other fact (not by any variables). You can suppress this with the `SAFE` flag. `jsonobjectinsert(SAFE $obj $key null)`.

ˆjsondelete(factid)

deprecated in favor of ˆdelete

ˆjsongather({fact-set} jsonid {level})

takes the facts involved in the json data (as returned by `ˆjsonparse` or `ˆjsonopen`) and stores them in the named factset.
This allows you to remove their transient flags or save them in the users permanent data file.

You can omit fact-set as an argument if you are using an assignment statement: `@1 = ˆjsongather(jsonid)`

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

ˆjsonlabel(label)

assigns a text sequence to add to jo- and ja- items created thereafter. See System functions manual.

ˆreadfile(TAB filepath {'ˆfn}) and ˆreadfile(LINE filepath'ˆfn)

reads a tsv (tab delimited spreadsheet file) and returns a JSON array representing it. The lines are all objects in an array. The line is an object where non-empty fields are given as field indexes. The first field is 0. Empty fields are skipped over and their number omitted.

If an optional 3rd parameter of a function name is given, the code does not create a JSON structure to return. Instead it calls the function with each field of a line being an argument. This is sort of analogous to `:document` mode in that you can potentially read large amounts of data in a single volley and may need to use `ˆmemorymark` and `ˆmemoryfree` to manage the issue.

`ˆreadfile ( LINE filepath 'function)` reads a file and passes each line untouched as the sole argument to the function.

Formerly called ˆjsonreadfile (still accepted).

An input line can be :trace all or :trace none to observe behavior of the data between them.

`^jsonundecodestring( string )`

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

WEB JSON

`^jsonopen( {JSONFLAGS} kind url postdata header {ignoreconcept {underscoreconcept} } {timeout})`

this function queries a website and returns a JSON datastructure as facts. It uses the standard CURL library, so it's arguments and how to use them are generally defined by CURL documentation and the website you intend to access. See writeup earlier about optional json flags.

parameter	description
kind	is `POST`, `GET`, `POSTU`, `GETU`, `PUT`, `DELETE` corresponding to the usual meanings of Get and Post and url-encoded forms.
url	is the url to query
postdata	is either "" if this is not a post or is the data to send as post or put
header	is any needed extra request headers or "". Multiple header entries must be separated by a tilde
timeout	optional seconds limitation for connection and then for transfer. else $cs_jsontimeout rules

Note: 'postdata' can be a simple JSON structure name, in which case the system will automatically perform a ^jsonwrite on it and send that text data as the data. Currently limited to 500K in size of the internal buffer.

A sample call might be:

```
$$url = "https://api.github.com/users/test/repos"
$$user_agent = ^"User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; WOW64; Tric
^jsonopen(GET $$url "" $$user_agent)
```

where GitHub requires user-agent data. Use `http://` and `https://` in front of your urls.

As an example of a complex header value you might create neatly,

```
$header = ^"Accept: application/json
~Accept-Encoding: identity,*;q=0
~Accept-Language: en-US,en;q=0.5
~Cache-Control: no-cache
~Connection: close
~Host: Chatscript
```

```
~User_Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:42.0) "
```

ChatScript will make each line have a single space separator between line continuations. And JsonOpen will correctly get the header elements that do not include that spacing.

The results are a collection of facts representing the JSON parse tree and the value of the function is the root JSON value. The JSON elements that can be represented are arrays, objects, JSON strings, and primitives (numbers, true, false, null). JSON arrays are named `ja-n` where `n` is a unique index. JSON objects are similarly named `jo-n`.

Unlike JSON, which makes a distinction between primitives and strings, in ChatScript those things are all strings and are not quoted. So a JSON string like this:

```
[ {"id": 1 "value": "hello"} {"id": 2 "value": "bye"} ]
```

returns this value: `ja-1` and these facts. The facts have flags on them which you can use in queries. You may not have any need to use these flags, so maybe you will just ignore their existence.

fact	associated flags
(ja-1 0 jo-1)	#JSON_ARRAY_FACT #JSON_OBJECT_VALUE
(jo-1 id 1)	#JSON_OBJECT_FACT #JSON_PRIMITIVE_VALUE
(jo-1 value hello)	#JSON_OBJECT_FACT #JSON_STRING_VALUE
(ja-1 1 jo-2)	#JSON_ARRAY_FACT #JSON_OBJECT_VALUE
(jo-2 id 2)	#JSON_OBJECT_FACT #JSON_PRIMITIVE_VALUE
(jo-2 value bye)	#JSON_OBJECT_FACT #JSON_STRING_VALUE

Using queries, you could get all values of an array. Or all fields of an object. Or all JSON facts where the field is the id. You could manually write script to walk the entire tree. But more likely you will use `^jsonpath` to retrieve specific pieces of data you want. For example, you could do a query on the value returned by `^jsonopen` as subject, to find out how many array elements there are. Then use `^jsonpath` to walk each array element to retrieve a specific field buried within.

Note - for things like primitive null, null arrays, null strings, null objects, these are represented as "null" and the corresponding fact flag tells you the kind of value it is.

You can also ask CS to show those out visually using `^jsontree`.

Note that the facts created are all transient and disappear at the end of the volley unless you have forced them to stay via `permanent`. Forcing them to stay is generally a bad idea because it will congest your user topic data file, slowing it down or exceeding its capacity, and because those facts may then collide with new facts created by a new `^jsonopen` on a new volley. The array and object

ids are cleared at each volley, so you will be reusing the same names on new unrelated facts.

Using the flag values, it is entirely possible to reconstruct the original JSON from the facts (if the root is an array or object because otherwise there are no facts involved), but I can't think of use cases at present where you might want to. You cannot compile CS on LINUX unless you have installed the CURL library. For Amazon machines that means doing this:

```
sudo yum -y install libcurl libcurl-devel
```

Or more completely, for evserver and curl, you might do:

```
sudo yum -y install -y libc -devel gcc libc6 -compat libcurl libcurl-devel libstdc++ libgcc
```

On some other machines that doesn't install library stuff and maybe you need

```
sudo apt-get install libcurl3 libcurl3-gnutls libcurl4-openssl-dev
```

System variables `%httpresponse` will hold the most recent http return code from calling `^jsonopen`.

If you call `^jsonopen(direct ...)` then the result will not be facts, but the text will be directly shipped back as the answer. Be wary of doing this if the result will be large (>30K?) since you will overflow your buffer without being checked.

^jsonopen automatically url-encodes headers and urls

Note: CS blocks until the call is complete. On a production server if you expect that the call can take serious time, you are advised to send the request in OOB to your own additional layer which returns to CS immediately. Then it makes the call and when the call completes, makes a call back to CS with the results in OOB.

You can set dynamic time limits on ^jsonopen. The variable $cs_jsontimeout sets a local seconds limit on the call. -1 means fail immediately.

If you have a variable $correlation_id, this value will be passed to jsonopen as an additional header correlation-id: %s. Used to track across multiple calls in a system what the originating message was.

If you have given an ignoreconcept, then whenver a field name is encountered in the incoming data matches the name of a member of that concept set, that field and all data below it are discarded and not returned.

If you have given an underscore concept, then any field name encountered in the incoming data that matches a member of that concept, that field's string value will have any spaces in it replaced with underscores.' I

JSONOpen and local files   JSONOpen is a wrapper around the `Curl` library, so you can use as an url a local file reference like `file:///c:/myfile.txt`. The

file, of course, should be a JSON text string.

JSONOpen and proxy servers If you need JSONOpen to run thru a proxy server, these are the CS variables you need to set up: `$cs_proxycredentials` should be your login data, e.g. `myname:thesecret`. `$cs_proxyserver` is the server address, e.g., `http://local.example.com:1080`. `$cs_proxymethod` are bits listing the authorization method to use. They come from the LIBCURL so you should OR together the bits you want. Bit 1 is the most basic choice of name and password. Read- https://curl.haxx.se/libcurl/c/CURLOPT_HTTPAUTH.html

JSON & Out-of-band output data

Out-of-band data in ChatScript is signaled by the output beginning with data enclosed in `[]`. Which might be confusing, since JSON uses `[]` to denote an array. Standalone ChatScript contains a built-in handler for OOB data and if you pass it JSON at the start of output, it will swallow it and not display it (unless you turn on OOB display).

Similarly, std webpage interfaces connecting to ChatScript do likewise. So if you want to see this information, you should put something in the output at the start which is NOT the JSON data. Anything will do. The only time you might actually need the JSON clean at the beginning is from some special purpose application, and in that case you will write your own OOB handler anyway (or not have one).

JSON & Out-of-band input data

OOB data into ChatScript is similarly signaled by being at the start of input, with data enclosed in [], followed typically by the user's actual input. The ChatScript engine reacts specially to OOB incoming data in that it will be careful to not treat it like ordinary user chat. Tokenization is done uniquely, spell-checking, pos-tagging, parsing, named entity merging etc are all turned off and the data becomes its own sentence (the user's actual input generates more sentences to CS as input). OOB data is then processed by your script in any way you want. So one clever thing you can do is pass in JSON data within the OOB to get temporary facts into your app during a volley. Input might look like this:

```
[ [ a b { "bob": 1, "suzy": 2 } ] ] What is your name?
```

You can pattern match the oob section of the input as follows:

```
u: ( \[ _* ) $_tmp = ^jsonparse('_0)
```

`_0` will contain an excess right bracket (the end of the oob message), but that won't bother `^jsonparse`.

Representing JSON in CS facts is more than just a bunch of subject-verb-object facts linked together.

The facts have typing bits on them that describe the structure and arrays have index values that must remain consistent. Therefore you should not create and alter JSON fact structures using ordinary CS fact routines like ˆcreatefact and ˆdelete. Instead use the JSON routines provided.

Practical Examples

Objects

The write routines jsonwrite and jsontree print out different views of the same data..

```
u: (-testcase1) $_jsonObject = ^jsoncreate(object)
    ^jsonobjectinsert( $_jsonObject name "some name" )
    ^jsonobjectinsert( $_jsonObject phone "some number" )
    ^jsonwrite ( $_jsonObject ) \n
    ^jsontree ( $_jsonObject )\n
```

Note in this next example how to escape a json string with ˆ". This makes creating json objects from static data very intuitive and clear.

```
u: (-testcase2) $_tmp = ^jsonparse( ^'{name: "Todd Kuebler", phone: "555.1212"}' )
    ^jsonwrite( $_tmp ) \n
    ^jsontree( $_tmp ) \n
    name: $_tmp.name, phone: $_tmp.phone
```

This example shows the . notation access of data inside a json object in chatscript. This is probably the most intuitive way of interacting with the data.

```
u: (-testcase3) $_tmp = ^jsoncreate(object)
    $_tmp.name = "Todd Kuebler"
    $_tmp.phone = "555-1212"
    ^jsonwrite( $_tmp ) \n
    ^jsontree( $_tmp ) \n
    name: $_tmp.name, phone: $_tmp.phone
```

Arrays of objects

In the example below, we add two items into an array of objects and we display the formatted array:

```
u: ( testcase4 )
    # create a phoneBook as an array of structured items (objects)
    $_phoneBook = ^jsoncreate(array)

    # add first object in the array
    #
```

```
$_item = ^jsoncreate(object)

# assign values
$_item.name = "Todd Kuebler"
$_item.phone = "555-1212"

^jsonarrayinsert($_phoneBook $_item)

# add a second object in the array
#
$_item = ^jsoncreate(object)

# assign values
$_item.name = "Giorgio Robino"
$_item.phone = "111-123456789"

^jsonarrayinsert($_phoneBook $_item)

# display JSON tree
^jsontree( $_phoneBook ) \n

#
# print formatted items in the phone book
#
phone book:\n
$_i = 0
$_size = ^length($_phoneBook)
loop($_size)
{
  # print out formatted item
  name: $_phoneBook[$_i].name, phone: $_phoneBook[$_i].phone\n
  $_i += 1
}
```