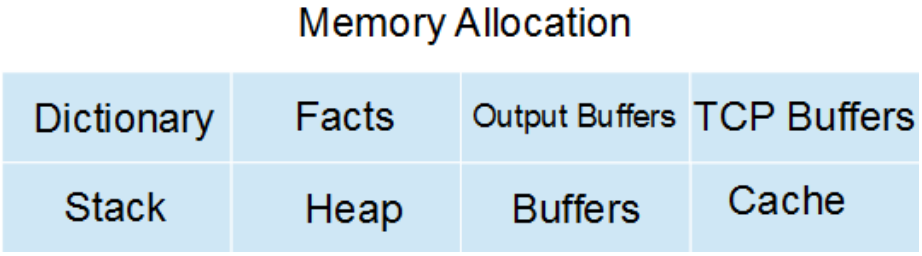
ChatScript Engine

- Data
- Memory Management
- Function Run-time Model
- Script Execution
- Evaluation Contexts
- Rule Tags
- Concept Representation
- Supplemental dictionary data
- Topic and rule representation
- Natural Language Pipeline
- Messaging
- Private Code
- Code Zones
- Documentation

This does not cover ChatScript the scripting language. It covers how the internals of the engine work and how to extend it with private code.

User Data

First you need to understand the basic data available to the user and CS and how it is represented and allocated.



Text Strings

The first fundamental datatype is the text string. This is a standard null-terminated C string represented in UTF8. Text strings represent words, phrases, things to say. They represent numbers (converted on the fly when needed to float and int64 values for computation). They represent the names of functions and concepts and topics and user variables. Strings are allocated out of either

the stack (transiently for $_ transient variables) or the heap (for normal user variables).

Dictionary Entries

The second fundamental datatype is the dictionary entry (typedef WORDENTRY). A dictionary entry points to a string in the heap and has other data attached. For example, function names in the dictionary tell you how many arguments they require, where to go to execute their code. User variable names in the dictionary store a pointer to their value string (in the heap). Topic names track which bots are allowed to use them. Ordinary English words have bits describing their part-of-speech information and how to use them in parsing.

In fact, ordinary words have multiple meanings which can vary in their part of speech and their ontology, so the list of possible meanings and descriptions is part of the dictionary entry. When working directly with a dictionary entry, the code uses `WORDP` as its datatype (pointer to a WORDENTRY). Code working indirectly with a dictionary entry uses `MEANING` as its datatype.

A `MEANING` is an index into dictionary space along with description bits that can say which meaning of the word is referred to (index into the meanings array of the dictionary entry) or which part of speech form (like noun vs verb) is being referred to. As a text string, a meaning can be either just the word (break) which represents all meanings. Or it can be a word with meaning index (break~33) which means the 33rd meaning of break. Or it can be a word with POS-tag (break~n) which means all noun meanings of "break". These meanings can be used in concepts and keywords in patterns.

Facts

The third fundamental datatype is the fact (typedef FACT), a triple of MEANINGs. The fields are called subject, verb, and object but that is just a naming convention and offers no restriction on their use.

Each field of a fact is either a `MEANING` (which in turn references a dictionary entry which points to a text string) or a direct index reference to another fact. As a direct reference, it is not text. It is literally an offset into fact space. But facts have description bits on them, and one such bit can say that a field is not a normal MEANING but is instead a binary number fact reference.

While fact references in a fact field are binary numbers, when stored in variables, a fact reference is a text number string like 250066. It's still an index into fact space. When stored in external files (like user long-term memory or `^export`) facts are stored as full text describing the fact, e.g.,

```
( bob like (apples and oranges) 0x01000000)
```

which in this case is a fact with a fact "(apples and oranges)" as object. Externally facts are stored this way because user facts get relocated every volley and cannot safely be referred to directly by a fact reference number. The binary number at the end is any flags that are describing this fact (in this case FACTTRANSIENT MEANING the fact dies at the end of the volley). And there may even be another number which tells what bots are allowed to know of this fact.

User Variables

User variables are names that can be found in the dictionary and have data (not their name) pointing to text string values. `$` and `$$` variables point to strings in the heap. `$_` variables point to strings in the stack.

Function Variables

Function variables like ^myvalue are defined in the arguments list of a function (outputmacro/patternmacro) and managed by the system outside of the dictionary. They handle pass-by-reference data for the duration of execution of a function. They are only visible within the code of that function.

Match Variables

Match variables (`_0`) hold text strings directly. They use prereserved spaces and thus have limited sizes of strings they can hold (MAX_MATCHVAR_SIZE / 20000 bytes). They are expecting to hold parts of a sentence that have been matched and sentences are limited to 254 words (and words are typically small). Any larger data is truncated upon storage.

A match variable is more complex than a user variable because a match variable holds extra data from an input sentence. It holds the original text seen in the sentence, the canonical form of that value, and the position reference for where in the sentence the data came from.

Match variables pass all that information along when assigned to other match variables, but lose all but one of them when assigned to a user variable or stored as the field of a fact.

Factset

Querying for relevant facts results in facts stored in an array called a factset, labeled @0, @1, etc. You treat these as arrays to retrieve a fact, normally specifying at the same time what part of the fact you want. But you can't directly specify the array index to retrieve from. You specify first, last, or next.

JSON Facts

CS directly supports JSON and you can manipulate JSON arrays and objects as you might expect. Internally, however, JSON is represented merely by facts with special bits that indicate how to use that JSON fact. JSON objects and arrays are represented as synthesized dictionary entries with names like like `jo-5` or `ja-1025` for permanent JSON and `ja-t5` for a transient one.

Each key-value pair of the object is a fact like `(jo-5 location Seattle)`. Each array element pair is a fact like `(ja-5 1 dog)`.

Array facts start with 0 as the verb and are always contigiously sequential (no missing numbers). Should you delete an element from the middle, all later elements automatically renumber to return to a contiguous sequence.

Variables holding JSON structures always just have the name of starting JSON name, like `jo-5` or `ja-t2`. That value, when you retrieve its corresponding dictionary entry, will have the JSON Facts bound to its cross-reference lists. Walking the structure will mean walking the lists of facts of that entry and subsequent JSON headers.

Memory Management

Many programs use malloc and free extensively upon demand. These functions are not particularly fast. And they lead to memory fragmentation, whereupon one might fail a malloc even though overall the space exists. ChatScript follows video game design principles and manages its own memory (except for 3rd party extensions like duktape and curl). It allocates everything in advance and then (with rare exception) it never dynamically allocates memory again, so it should not fail by calling the OS for memory. You have control over the allocations upon startup via command line parameters.

This does not mean CS has a perfect memory management system. Merely that it is extremely fast. It is based on mark/release, so it allocates space rapidly, and at the end of the volley, it releases all the space it used back into its own pool. In the diagram below under Memory Allocation you have a list of all the areas of memory that are preallocated.

You might run out of memory allocated to dictionary items while still having memory available for facts. This means you need to rebalance your allocations. But most people never run into these problems unless they are on mobile versions of CS.

Stack and Heap memory are allocated out of a single chunk. Stack is at the low end of memory and grows upwards while Heap is at the high end and grows downward. If they meet, you are out of space.

Stack memory is tied to calls to `ChangeDepth` which typically represent function or topic invocation. Once that invocation is complete, all stack space allocated is cut back to its starting position. Stack space is typically allocated by `AllocateStack` and explicitly released via `ReleaseStack`. If you need an allocation but won't know how much is needed until after you have filled the space, you can use InfiniteStack and then when finished, use `CompleteBindStack` if you need to keep the allocation or `ReleaseInfiniteStack` if you don't. While `InfiniteStack` is in progress, you cannot safely make any more `AllocateStack` or `InfiniteStack` calls.

Heap memory allocated by `AllocateHeap` lasts for the duration of the volley and is not explicitly deallocated. So conceivably some heap memory is free but hasn't been freed until the end of the volley.

Fact memory is consumed by `CreateFact` and lasts for the duration of the volley.

But CS supports planning, which means backtracking, which means memory is really not free along the way because the system might revert things back to some earlier state. This problem of free memory mostly shows up in document mode, where reading long paragraphs of text are all considered a single volley and therefore one might run out of memory. CS provides `^memorymark` and `^memoryfree` so you can explicitly control this while reading a document. And more recently provides `memorygc` which copies data from heap to stack, and then copies back only the currently used data.

Buffers are allocated and deallocated via `AllocateBuffer` and `FreeBuffer`. Typically they are used within a small block of short-lasting code, when you don't want to waste heap space and cannot make use of stack space. While there are a small amount of them preallocated, in an emergency if the system runs out of them it can malloc a few more.

TCP buffers are dynamically allocated (violating the principle of not using malloc/free) via accept threads of a server.

Output buffers refer to either the main user output buffer or the log buffer. The output buffer needs to be particularly big to hold potentially large amounts of OOB (out-of-band) data being shipped externally.

Also most temporary computations from functions and rules are dumped into the output buffer temporarily, so the buffer holds both output in progress as well as temporary computation. So if your output were to actually be close to the real size of the buffer, you would probably need to make the buffer bigger to allow room for transient computation. The log buffer typically is the same size so one can record exactly what the server said. Otherwise it can be much smaller if you don't care.

Cache space is where the system reads and writes the user's long term memory. There is at least 1, to hold the current user, but you can optimize read/write calls by caching multiple users at a time, subject to the risk that the server

crashes and recent transactions are lost. A cache entry needs to be large enough to hold all the data on a user you may want saved.

Layers

When ChatScript starts up, it loads data in layers. First is all the permanent data associated with the system, which is printed out as `WordNet:` and loads dictionary entries and facts. It generates some concepts and loads LIVEDATA information.

Then the system loads data the TOPIC folder. `Build0:` and then `Build1:`. These contain facts, more dictionary data, concept sets, variables with values, and scripted function definitions.

Then if there is a boot function, it executes and additional data can be brought into the boot layer.

The system is now ready for user inputs. User folder data has a topic file to represent a user talking to a specific bot. That data is read into the user layer. The input volley alters content of the layer and outputs messages to the user. Changes in facts and user variables as well as execution state of topics are stored back into the user's topic file. Then the user layer is removed and the system is ready for another user.

All layers are "unwindable", like peeling an onion. And all layers are protected from harm by later layers. However, it is possible to create facts (and hence dictionary entries) in the user layer and migrate some of them into the boot layer when the user volley is finished. This means that data will be globally visible in the future to all users.

When you create a new fact, the corresponding dictionary entry for each field either already exists or is newly created. The entry has lists for each field it participates in and the fact is added to the head of that list. This coupling needs to be unwound when we want to remove a layer. When we want to unwind, we walk facts from most recently created to start of the layer. By going to the dictionary entry of a field, that fact will be the current first entry of the list, so we merely pop it from the list and that decouples fact and dictionary entry. Once decoupled, all newly created dictionary entries are no longer needed so the free entry pointer can be reset to the start of the layer. And all user layer facts have been written out to a file, so we can reset the free fact pointer as well. Similarly all factsets that need saving were written out, so they can all be reset empty. User Variables will also have been written out, so they too can have their values cleared so they are no longer pointing into the heap. Nothing from the volley will now be occupying any heap memory and so its pointer can be reset back to start of layer also.

Garbage collection

Using the layer model, normally user conversation causes heap allocations and dictionary allocations that can be rapidly released when the user layer is unwound. And plenty of space should exist to manage without any garbage collection occurring within the volley. But particularly if you are in document-reading mode, processing all sentences of a book happen within a single volley. CS does not have the memory for that without some form of garbage collection. There are two such mechanisms built into the engine.

The simplest gc is a user-controllable mark-release. You find a safe place in script to use mark, you process a sentence, and then you use release to restored dictionary and heap values back to the mark. This is fine if your results are going to An external place (like writing to a file with ^log). But it is hard to keep any results around past the release call. In fact, the only resident way to do that is to write something onto a match variable. Match variables have their own memory and last until changed. They are a way to pass information between different user volleys or, in this case, past a mark-release barrier.

The complex gc is an actual gc, where data is moved around and space gets recompacted after discarding trash. Normally facts consist of MEANINGS, and those dictionary entries know what facts refer to them. So things can be shuffled around. EXCEPT...

Facts don't know what factsets they may be listed in, and facts that later facts directly refer to by index are unaware of it. And JSONLOOP is maintaining facts it is using and will be unaware if they are relocated. And if you request a fact id from some query result, that id is really only good for the duration of the volley, and not safe across a gc because it will be stored on a variable which has no cross-reference from the fact. So care is required to use the gc mechanism.

Finer details about words

Words are the fundamental unit of information in CS. The original words came from WordNet, and then were either reduced or expanded. Word are reduced when some or all meanings of them are removed because they are too difficult to manage. `I`, for example, has a Wordnet meaning of the chemical iodine, and because that is so rare in usage and causes major headaches for ChatScript (noun instead of pronoun), that definition has been expunged along with some 500 other meanings of words.

Additional words have been added, including things that Wordnet doesn't cover like pronouns, prepositions, determiners, and conjunctions. And more recent words like `animatronic` and `beatbox`. Every word in a pattern has a value in the dictionary. Even things that are not words, including phrases, can reside in the dictionary and have properties, even if the property is merely that this is a keyword of some pattern or concept somewhere.

Words have zillions of bits representing language properties of the word (well, maybe not zillions, but 3x64 bytes worth of bits). Many are permanent core properties like it can be a noun, a singular noun, it refers to a unit of time (like `month`), it refers to an animate being, it's a word learning typically in first grade.

Other properties result from compiling your script (this word is found in a pattern somewhere in your script). All of these properties could have been represented as facts, but it would have been inefficient in either cpu time or memory to have done so.

Some dictionary items are `permanent`, meaning they are loaded when the system starts up, either from the dictionary or from data in layer 0 and layer 1. Other dictionary items are `transient`. They come into existence as a result of user input or script execution and will disappear when that volley is complete. They may live on in text as data stored in the user's topic file and will reappear again during the next volley when the user data is reloaded. Words like dogs are not in the permanent dictionary but will get created as transient entries if they show up in the user's input.

The dictionary consists of `WORDENTRY`s, stored in hash buckets when the system starts up. The hash code is the same for lower and upper case words, but upper case adds 1 to the bucket it stores in. This makes it easy to perform lookups where we are uncertain of the proper casing (which is common because casing in user input is unreliable). The system can store multiple ways of upper-casing a word.

Facts are simply triples of words that represent relationships between words. The ontology structure of CS is represented as facts (which allows them to be queried). Words are hierarchically linked (WordNet's ontology) using facts (using the `is` verb). Words are conceptually linked (defined in a `~concept` or as keywords of a `topic`) using facts with the verb `member`.

Word entries have lists of facts that use them as either subject or verb or object so that when you do a query like

```
^query(direct_ss dog love ?)
```

CS will retrieve the list of facts that have dog as a subject and consider those. And all those values of fields of a fact are words in the dictionary so that they will be able to be queried.

Queries like

```
^query(direct_v ? walk ?)
```

function by having a byte code scripting language stored on the query name `direct_v`. This byte code is defined in `LIVEDATA` (so you can define new queries) and is executed to perform the query. Effectively facts create graphs and queries are a language for walking the edges of the graph.

ChatScript supports user variables, for considerations of efficiency and ease of

reference by scripters. Variables could have been represented as facts, but it would have increased processing speed, local memory, and user file sizes, not to mention made scripts harder to read.

Function Run-time Model

The fundamental units of computation in ChatScript are functions (system functions and user outputmacros) and rules of topics. Rules and outputmacros can be considered somewhat interchangeable as both can have code and be invoked (rules by calling `^reuse`). And both can use pattern matching on the input.

Function names are stored in the dictionary and either point to script to execute or engine code to call, as well as the number of arguments and names of arguments.

System functions are predefined C code to perform some activity most of which take arguments that are evaluated in advance (but use `STREAMARG` and the function waits until it gets them to decide whether to evaluate or not). System functions can either designate exactly how many arguments they expect, or use `VARIABLE_ARGUMENT_COUNT` to allow unfixed amounts.

Outputmacros are scripter-written stuff that CS dynamically processes at execution time to treat as a mixture of script statements and user output words. They can have arguments passed to them as either call by value or call by reference. The scripter functions are loaded from `macros0.txt` and `macros1.txt` In the TOPIC folder. Functions are stored 1 per line, as compilation goes along, so more than one line may define the same function.

```
^car_reference o 2048 0 A( ) ...compiled script...
```

The name (^car_reference) is followed by the kind of function (outputmacro, tablemacro, patternmacro, dualmacro), followed by the bot bits allowed to use this function (2048 bot), followed by flags on the function (0), followed by the number of arguments. It expects (0). Argument count is 'A' + count when count <= 15 and 'a' + count - 15 when greater.

The value of the dictionary entry of the function name is a pointer to the function data allocated in the heap. Multiple definitions of the name are chained together, and the system will hunt that list for the first entry whose bot bits allow use.

Patternmacros are scripter-written patterns that allow some existing pattern to transfer over to them and back again when used up. They have the same format as outputmacros, but their code data is simply the pattern to execute. The pattern code merely switches over to this extension code until it runs out, and resumes its normal code. You can't nest patternmacro calls at present.

Argument Passing

There are actually two styles of passing arguments. Arguments are stored in a global argument array, referenced by ARGUMENT(n) when viewed from system routines. The call sets the current global index and then stores arguments relative to that. Outputmacros use that same mechanism for call-by-reference arguments. Call by reference arguments start with ^ like `^myarg` and the script compiler compiles the names into number references starting with `^0` and increasing `^1` ... `^myarg` style allows a routine to assign indirectly to the callers variable.

But outputmacros also support call by value arguments which start with `$_` like `$_myarg`. No one outside the routine is allowed to change these or use these to access the above caller's data. Hence call by value. These are normal albeit transient variables so the corresponding value from the argument stack is also stored as the value of the local variable. That happens after the function call code first saves away the old values of all locals of a routine (or topic) and then initializes all locals to NULL. Once the call is finished, the saved values are restored.

When passing data as call by value, the value stored always has a backtick-backtick prefix in front of it. In fact, all assignments onto local variables have that prefix prepended (hidden). This allows the system to detect that the value comes from a local variable or an active string and has already been evaluated. Normally, if the output processor sees `$xxx` in the output stream, it would attempt to evaluate it. But if it looks and sees there is a hidden back-tick back-tick before it, it knows that `$xxx` is the final value and is not to be evaluated further.

Back-tick (‘) is a strongly reserved character of the engine and is prevented from occurring in normal data from a user. It is used to mark data coming preevaluated. It is used to mark ends of rules in scripts. It is used to quote values of variables and fact fields when writing out to the user's topic file. It is used to create specific dictionary entries that cannot collide with normal words.

Script Execution

The engine is heavily dependent upon the prefix character of a script token to tell the system how to process script. The script compiler normally forces separate of things into separate tokens to allow fast uniform handling. E.g., `^call(bob hello)` becomes `^call ( bob hello )`.

This predictability allows the system to avoid all the logic involved in knowing where some tokens end and others begin. The other trick the script compiler uses is to put in characters indicating how far something extends. This jump value is used for things like if statements to skip over failing segments of the if.

Actual script execution, be it output processing or pattern processing jumps via switch statements on the initial character of a token.

Rule Tags

CS executes rules. While scripters can add their own label to a rule, all rules are automatically labelled internally by their position in a topic. The topic has a list of rules, numbered from 0 ... for top level rules. Top level rules can have rejoinders, which are numbered from 1 ... The system creates a text rule tag like: ~books.12.0 which means in the topic books, the 13th top level rule, and at the top level (not a rejoinder). ~books.12.3 is the third rejoinder under top-level rule 12. Internally a rule id has the bottom 16 bits for top level id and the next 16 for the rejoinder id.

Engine function arguments involving rules can accept either user labels or rule tags. While rule tags are completely unique, nothing prevents a user from labelling multiple rules in a topic with the same label, which is sometimes useful (eg in ^reuse for finding a rule not yet disabled or in ^incontext).

Concept representation

A concept is a word beginning with ~. Ideally it has a bit on it that tells us that it is an officially compiled concept or topic (topics are also concepts via their keywords list). Members of the concept are facts whose verb is Member and whose object is the concept name. Since these facts are stored as references from the concept name in object field position, all members can be found (including ones merely defined by `^createfact(myname member ~someconcept))`.

In the TOPIC folder the files `keywords0.txt` and `keywords1.txt`. Since concepts are represented as facts, the data needed is the name of the concept/topic, the list of keywords to create into facts upon loading is provided, along with the bot bits that identify which bots can see that fact. Below's first entry is a topic (T~) and the second is a concept. The third is a concept with concept flags and because it comes from a multi-bot environment so each word is joined to the botbits that can see that keyword.

```
T~introductions ( here name ~emohowzit ~emohello ~emogoodbye )
~introductions ( here 'name ~emohowzit ~emohello ~emogoodbye )
~black PROBABLE_ADJECTIVE ( dead`16 dark`16 blank`16 )
```

An apostrophe in front of the word means only that word and not any conjugations of it.

Concepts also allow patterns as members, but they are not saved as MEMBER facts. They are saved as conceptpattern facts, where the subject is the pattern

and the object is the concept name. They are executed after the normal NL pipeline is complete. The pattern can be compiled or uncompiled. Coming from compiling script they are compiled. Coming from a function like ^testpattern they may or may not be compiled. If uncompiled, the system will compile them first, every volley.

Concepts can contain other concepts as members. When you build a concept by including another concept, you can also elect to exclude specific subconcepts or words. In the representation for this, all normal concept facts are last in the list and exclusions are first. So a word that wants to trigger a concept must first pass against the exclusions list. Set exclusions are in the middle, while the start of the members are the single words that must be excluded. Therefore in the marking phase, when we have a word and we are chasing up what concepts its a member of, if the word is in the simple excludes list, we dont continue marking That set with it. If the exclusions are sets, we have to defer decision making until other paths have been chased up, to see if the set has been marked.

Supplemental dictionary data

The TOPIC files `canon0.txt` and `canon1.txt` hold results of compiling the canon: declaration. Each line is a pair of words, the original and what it's canonical should be. Similarly `private0.txt` and `private1.txt` files hold pairs spell-check replacements, though they can handle multiple word in and multiple word out.

The files `dict0.txt` and `dict1.txt` handle words whose property bits have been created or changed by compilation.

```
+ Expedition_Limo. NOUN NOUN_PROPER_SINGULAR
```

In the example above the word underwent positive change with addition of NOUN and NOUN_PROPER_SINGULAR property bits.

Topic and rule representation

Topic keywords are stored along with concept keywords in the keywordsn.txt files. The files `topic0.txt` and `topic1.txt` contain the rest of topic data. Each entry consists of 2 lines. The first tells you its a topic, names it, and names various flags and properties about it including where it is defined in source. The second line first lists all botnames allowed to use this topic and then begins listing all the rules.

```
TOPIC: ~chatbots 0x1b 53459696 34 0 2798 chatbots.top
"  mybot  " 012 ?: ( do * you * ~like * robot ) Robots are cool. `
```

Rules start with a jump index to the next rule(012), have their rule type(?:), pattern, and output. A rule ends with the backtick mark.

If there are multiple copies of the topic (due to multiple bots), they are just another line pair show botname list will be different.

Evaluation Contexts

The evaluation contexts are

```
outputSystem.cpp output function
performAssignment left side
if testing
activeString ReformatString
all STREAM argument engine functions
DoFunction function call argument evaluation
```

Many things evaluate the same way. Match variables (`_0`) evaluate to their content. User variables evaluate to their content, unless they are dotted or subscripted, in which case they evaluate to their content and then perform a JSON object lookup.

Messaging

CS user messaging design involves several features: directness, cancelability, and accountability.

Directness:

Normal computer languages have computation as the main goal, with user output as a last result. But a chatbot language has user output as the main goal and may or may not ever involve computation. Since ChatScript strives to make it easy to create chatbots, it heads in a different direction from Normal languages. A normal language might output to the user like this:

```
printf("The result is %d for %s units.", value, units);
```

But this involves typing extraneous function calls and punctuation and creating dummy values and putting real values later, where errors in order or count might occur. In CS this would be:

```
The result is $value for $units.
```

Cancelability:

CS is evaluating tokens for output to the user as it goes along. This may mean processing words, or variables, or functions. But a function might fail, and we

don't want to send a partial output to the user. So what we do is
put tokens into a transient output stream. If at any time we "fail", then we simply discard the stream. If, however, we succeed, then the stream is transferred into a message unit to be shipped to the user later when we are done.

Accountability:

Each message unit saves the message (typically one or more sentences) and the rule that generated it. In order to have this accountability, no message unit can cross multiple rules (though a rule may choose to generate multiple message units). Because we have the name of the rule involved, we can later know why the message arose (with good rule naming) and even decide to revise or delete the message entirely.

To achieve this, whenver a rule completes, it transfers any output from the transient output stream into being a message unit. Furthermore, if a rule would transfer control to another rule, it transfers what it has so far into a message unit. It may generate more transient output after it returns from whatever rule it invoked. Rule transfers happen with: ^gambit, ^respond, ^refine, ^sequence, ^reuse. ^retry, while not triggering a new rule, re-triggers this rule and causes this behavior, as does ^print which is an explicit request to generate a message unit. Also ^postprintbefore and ^postprintafter.

While there may be many message units to show to the user, the result is merely to concatenate them with space separators. So from the user's view there is no visibility over "message units", there is just the resulting message.

Natural Language Pipeline

CS pre-processes user input to make it easy to find meaning within it. These are both classic and unusual NL processing steps. All steps in the pipeline use data representations that work together. Much of the work aims toward normalization, that is making different ways of typing in the same thing look the same to scripts, so they don't have to account for variations (making script writing and maintenance easier). Other than tokenization, all other pipeline steps are under control the script (`$cs_token`), which can choose to use any combination of them at any time.

Tokenization

Tokenization is the process of taking a stream of input characters and breaking it sentences consisting of words and punctuation (tokens).

ChatScript can process any number of sentences as a single input, but it will do so one at a time. In addition to the naive tokenization done by other systems, CS also performs some normalizations at this point.

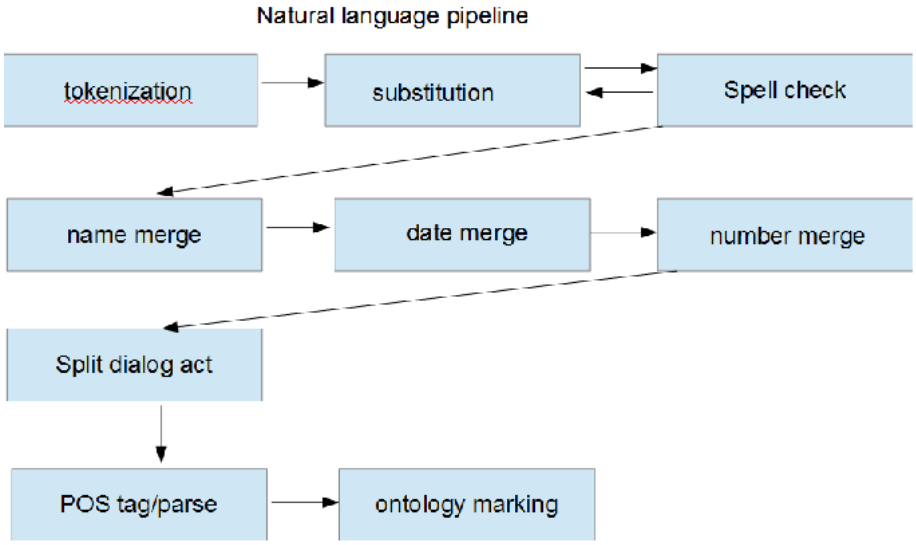


Figure 1: ChatScript NLP pipeline

This is also where out-of-band (OOB) data gets split off from user input. ChatScript can send and receive information from the user and the application simultaneously.

Inbound OOB can pass along any context like what category and specialty the user is starting in or what sensors have seen of the user (gesture recognition, etc).

Outbound OOB can tell the system how to manipulate an avatar, why the current outgoing user message was emitted, or any other special system data. The CS convention for OOB is that it is always first in the message and is encased in `[ ]`.

The input is a stream of characters. If there is OOB data, this is detected and split off as a single sentence in its own right. The rest of the input is then separated into the next sentence and the leftover unprocessed characters. A sentence has a 254 word limit, so anything attempting to be one beyond that will be split at that boundary.

Tokenization tries to separate into normal words and special tokens, e.g. `I like (this)`. tokenizes into `I like ( this )` . It has a bunch of decisions to make around things like periods and commas. Is the period a sentence end or an abbreviation or web url or what? Is a comma a part of a number or a piece of punctuation of the sentence? If something is a `12/2/1933` date, it can be separated into `12 / 2 / 1933` which leaves the decision to date merge or not under control of the scripter.

The process of tokenization is not visible to the script. It cannot easily know what transformations were made. This becomes the real sentence the user input, and is visible from the ^original function. You can see the result using `:tokenize some sentence`.

Substitution

Substitution involves replacing specific tokens or token sequences with different ones. Substitutions come from the LIVEDATA folder or `replace:` in scripts. A substitution represents sequences of words one might expect and how you might want to revise them. Revisions are either to correct the user input or make it easier to understand the meaning by converting to a standard form. The files include:

- british: a british spelling and its american equivalent like *colour* into *color*
- contractions: expands contractions like *don't* into *do not*
- interjections: revises dialog acts like so long. into `~emogoodbye` or *sure.* into `~yes`
- noise: removes useless words like *very* or *I mean that*
- spellfix: common spelling mistakes that might be hard for spellcheck to fix correctly
- substitutes: for whatever reason - e.g. Britain into Great Britain
- texting: revise shorthands into normal words or interjections like *:)* into `~emohappy`

The format is typically original word followed by replacement. E.g.

```
switc  switch
```

To represent multiple words to handle on input, separate them with underscores or place in double quotes. To indicate the word must be a sentence start use < and a sentence end use >. To represent multiple words on output, use + between the words. An underscore in output means issue the data as you see it, a composite word with underscore between. This is how Wordnet represents multiple word entries.

```
<real_estate> my+holdings
```

The above detects the two word sentence `real estate` and converts it into two different words `my holdings`

Spell check

Standard spell check algorithms based on edit distance and cost are used, but only among words which are of the same length plus or minus one.

The system also checks for bursting tokens into multiple words or merging multiple tokens into a word, and makes decisions about hypenated words.

For English, since the dictionary only contains lemma (canonical) forms of words, it checks standard conjugations to see if it recognizes a word.

For foreign languages, the engine lacks the ability to conjugate words, so the dictionary needs to include all conjugated forms of a word.

Merging

Merging converts multiple tokens into single ones that can be manipulated more easily. One can have proper names merged (*John Smith* → *John_Smith*) which supports the classic named-entity extraction (finding proper names). Date tokens merged (*January 2, 1990* → *January_2,1990*), and number tokens merged.

Number merging, in particular, converts things like *four score and seven years ago* into a single token *four_score_and_seven_years_ago* which is marked as a number and whose canonical value is *87*.

Splitting

Dialog acts can be split into separate sentences so that *Yes I love you* and *Yes, I love you* and *Yes. I love you* all become the same input of two sentences - the dialog act `~yes` and *I love you*.

Run together words may be split if the composite is not known and the pieces are.

Pos-parsing

Pos-Parsing performs classic part-of-speech tagging of the tokens. This means that *He flies* where *flies* is a verb in present 3rd person is distinguished from *He eats flies* where *flies* is a plural noun.

The system also attempts to parse the sentence to determine things like what is the main subject, main verb, main object, object of a clause or phrase, etc.

For English, which is native to CS, the system runs pos-parsing in two passes. The first pass is execution of rule from `LIVEDATA/ENGLISH/POS` which help it prune out possible meanings of words. The goal of these rules is to reduce ambiguity without ever throwing out actual possible pos values while reducing incorrect meanings as much as possible.

The second pass tries to determine the parse of the sentence, forcing various pos choices as it goes and altering them if it finds it has made a mistake. It uses a `garden path` algorithm. It presumes the words form a sentence, and tries to

directly find pos values that make it so in a simple way, changing things if it discovers anomalies.

For foreign languages, the system has code that allows you to plug in as a script call things that could connect to web-api pos-taggers. It also can directly integrate with the TreeTagger pos-tagger if you obtain a commerical license for one or more languages from them. Parsing is not done by TreeTagger so while you know part-of-speech data, you don't know roles like mainsubject, mainverb, etc. But some languages come with chunking, which you can also use to mark chunks as concepts.

Ontology Marking

Ontology Marking performs a step unique to ChatScript. It marks each word with what alternative views one might have of it. Pattern matching can match not just specific words but any of the alternate views of a word.

A pattern like `(I * ~like * ~animals)` can match any sentence which has that rough meaning, covering thousands of animals and dozens of words that mean to like. These `~` words are concept names, and ChatScript can match them just as easily as it can match words.

CS finds concept sets a word belongs to. Concept sets are lists of words and phrases (and concepts) where the words have some kind of useful relationship to each other.

A classic concept set is a synonym of a word. `~like` is the set of words that mean to like, e.g., *admire*, *love*, *like*, *take a shine to*, etc.

Another kind of concept set is a property of things like `~burnable` which lists substances and items that burn readily.

A third concept set kind defines affiliated words, like `~baseball` has *umpire*, *batv*, *ball*, *glove*, *field*, etc. And yet another concept set can define similar objects that are not synonyms, like `~role` which is the set of all known human occupations.

ChatScript comes with 2000 such sets, and it is easy for developers to create new ones at any time.

Pos-tags like `~noun`, `~noun_singular`, and sentence roles like `~mainSubject` are also concept sets and so the results of pos-tagging merely become marked concept sets attached to a word.

Marking also does the classic lemmatization (finding the canonical root). This includes canonical numbers so a number-merged *one thousand three hundred and two* which became a single token has the canonical form *1302* also.

Marking means taking the words of the sentence in order (where they may have pos-specific values) and noting on each word where they occur in the sentence (they may occur more than once).

From specific words the system follows the member links to concepts they are members of, and marks those concepts as occuring at that location in the sentence. It also follows is links of the dictionary to determine other words and concepts to mark. And concepts may be members of other concepts, and so on up the hierarchy. There exist system functions that allow you, from script, to also mark and unmark words. This allows you to correct or augment meanings.

In addition to marking words, the system generates sequences of 5 contiguous words (phrases), and if it finds them in the dictionary, they too are marked.

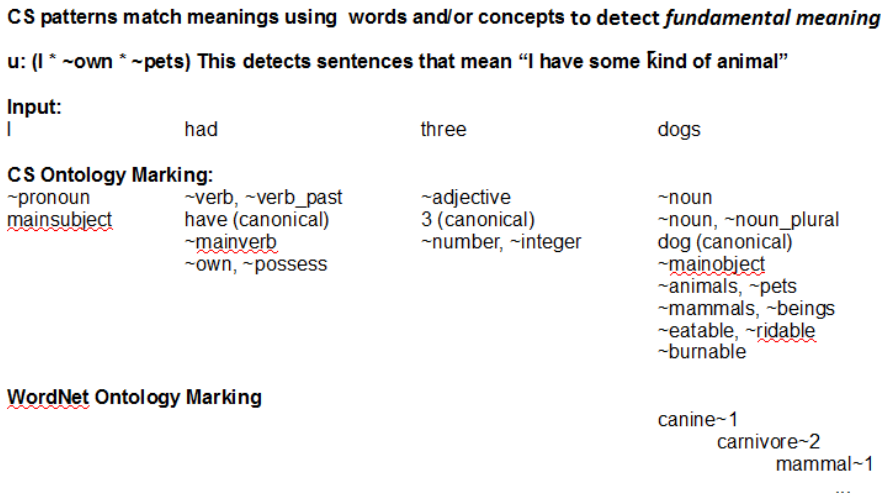


Figure 2: ChatScript Ontology

Script Compiler

In large measure what the compiler does is verify the legality of your script and smooth out the tokens so there is a clean single space between each token. In addition, it inserts `jump` data that allows it to quickly move from one rule to another, and from an `if` test to the start of each branch so if the test fails, it doesn't have to read all the code involved in the failing branch.

It also sometimes inserts a character at the start of a patttern element to identify what kind of element it is. E.g., an equal sign before a comparison token or an asterisk before a word that has wildcard spelling.

Private Code

You can add code to the engine without modifying its source files directly. To do this, you create a directory called privatecode at the top level of ChatScript. You must enable the `PRIVATE_CODE` define.

Inside it you place files:

`privatesrc.cpp`: code you want to add to functionexecute.cpp (your own cs engine functions) classic definitions compatible with invocation from script look like this:

```
static FunctionResult Yourfunction(char* buffer)
```

where ARGUMENT(1) is a first argument passed in. answers are returned as text in buffer, and success/failure codes as FunctionResult.

`privatetable.cpp`: listing of the functions made visible to CS table entries to connect your functions to script:

```
{ (char*) ^YourFunction, YourFunction, 1,0, (char*) help text of your function},
```

`1` is the number of evaluated arguments to be passed in where `VARIABLE_ARGUMENT_COUNT` means args evaled but you have to detect the end and `ARGUMENT(n)` will be `?`

Another possible value is `STREAM_ARG` which means raw text sent. You have to break it apart and do whatever.

`privatesrc.h`: header file. It must at least declare:

```
void PrivateInit(char* params);  called on startup of CS, passed param: private=
void PrivateRestart(); called when CS is restarting
void PrivateShutdown();   called when CS is exiting.
privatetestingtable.cpp  listing of :debug functions made visible to CS
```

Debug table entries like this:

```
{(char*) ":endinfo", EndInfo,(char*)"Display all end information"},
```

Code Zones

The system is divided into the code zones shown below. All code is in SRC.

Core Engine

- `dictionary` is `dictionary`
- `facts/JSON queries` is `factSystem`, `json`, `jsmn`, and * * * `infer`
- `long term user memory` is `userCache`, `userSystem`

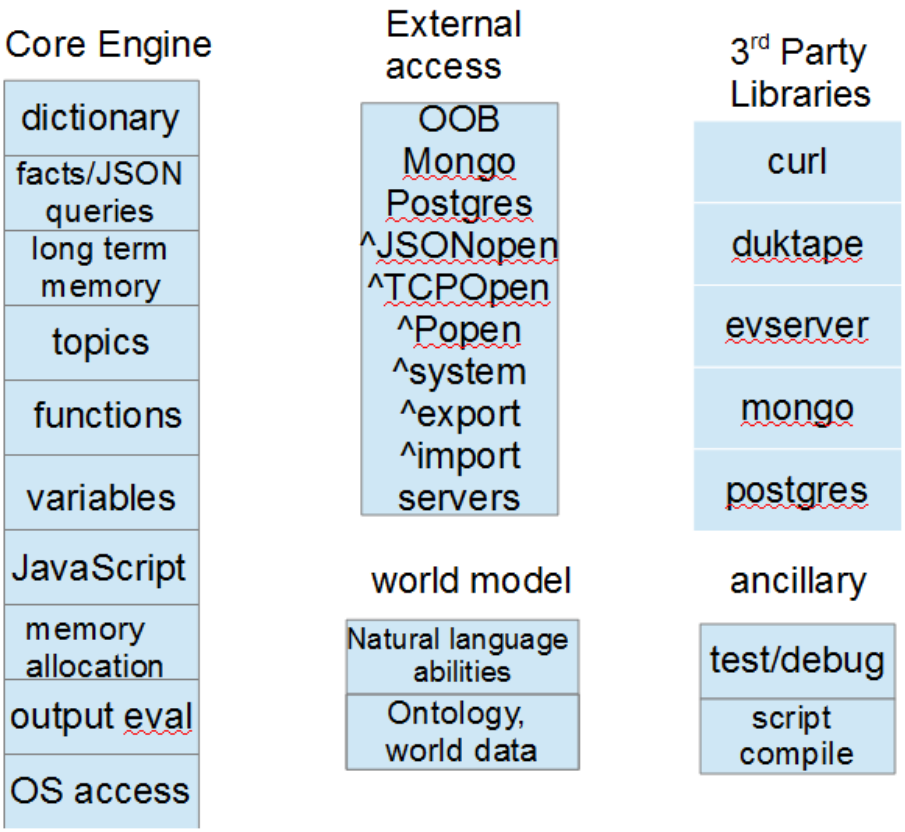


Figure 3: ChatScript architecture

- `topics` and loading the results of compilation is `topicSystem`
- `functions` is `functionExecute`
- `variables` is `variableSystem` and `systemVariables`
- `JavaScript` is `javascript`
- `memory allocation` and other things like file system access is in `os`
- `output eval` is `outputSystem`
- folders inside SRC are external systems included in CS, including: “‘
- curl (web api handling)
- duktape (javascript evalutation)
- mongo (mongodb access)
- postgres (postgres access)
- evserver (LINUX fast server) “‘
- Code to handle if and loop are in `constructCode`
- Miscellaneous text processing abilities are in `textUtilities`.

External Access

- `OOB` is partly in `tokenSystem` and mostly in `mainSystem`
- `Mongo` is `mongodb`
- `Postgres` is `postgres`
- All of the system's `^functions` are in `functionExecute`
- Servers are `evserver`, `cs_ev` and `csocket`

World Model

- Natural language abilities are in `english`, `englishTagger`, `markSystem`, `patternSystem`, `spellcheck`, `tagger` and LIVEDATA and DICT
- RAWDATA/ONTOLOGY contains ontology data
- RAWDATA/WORLDDATA contains world data

3rd Party Libraries

- curl manages web protocols for ^JSONOpen
- duktape is a JavaScript interpreter
- LINUX evserver allows multiple copies of CS tied to port
- Mongo connects to a remote MongoDB

- Postgres connects to a remote Postgres db
- jsmn is direct source that parses JSON

Ancillary

- Test and Debug is in `testing`
- Script compiler is in `scriptCompile`

Documentation

Master documentation is in the `WIKI` folder, with `PDFDOCUMENTATION` and `HTMLDOCUMENTATION` generated from that using scripts in `WIKI`.