ChatScript Engine

- Code Zones
- Data
- Memory Management
- Function Run-time Model
- Script Execution
- Rule Tags
- Concept Representation
- Pattern Matching (ChatScript-Engine-md#pattern-matching)
- Supplemental dictionary data
- Topic and rule representation
- Natural Language Pipeline
- Messaging
- Error Handling
- Multiple Bots
- Private Code
- 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.

Code Zones

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

Core Engine

- `startup and overall control` is `mainSystem`
- `dictionary` is `dictionarySystem`
- `facts/JSON queries` is `factSystem`, `json`, `jsmn`, and `infer`
- `topics` and loading the results of compilation is `topicSystem`
- `functions` is `functionExecute`
- `variables` is `variableSystem` and `systemVariables`
- `memory allocation` and other things like file system access is in `os`
- `output eval` is `outputSystem`
- `system variable access` is `systemVariables`
- `user variable access` is `variableSystem`
- `pattern matching` is `patternSystem`
- `text functions` is `textUtilities`
- `concept marking` is `markSystem`

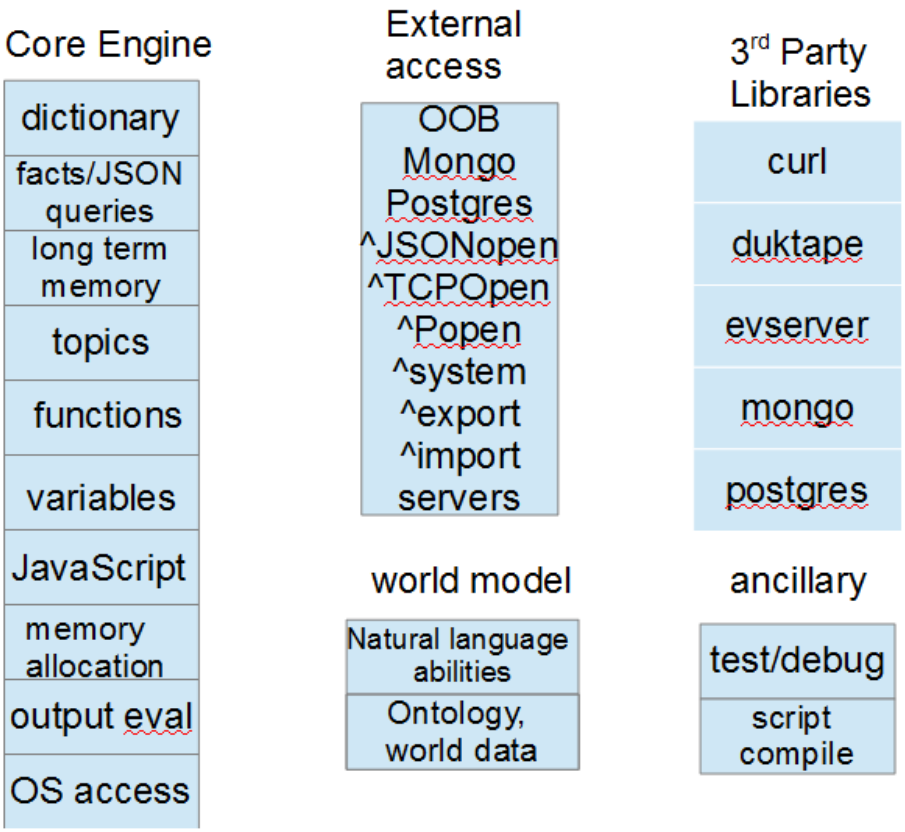

Figure 1: ChatScript architecture

- `if and loop` are in `constructCode`

Natural Language stuff

- `tokenization` is `spellcheck` and `tokenSystem`
- `English pos-tagging and parsing` is `english` and `englishtagger` and `tagger`

Statefulness

- `long term user memory` is `userCache`, `userSystem`

Servers

- `server` abilities are in `csocket` (Windows) and `evserver/cs_ev` (Linux)

JavaScript

- `JavaScript` is in `javascript`

Databases

- `Mongo access` is `mongodb`
- `Postgres access` is `postgres`

Ancillary

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

Folders inside SRC are external systems included in CS, including:

- `curl` (web api handling/protocols for ^JSONOpen)
- `duktape` (JavaScript evaluation)
- `evserver` (LINUX fast server)
- `mongo` (mongodb access)
- `postgres` (postgres access)

User Data

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

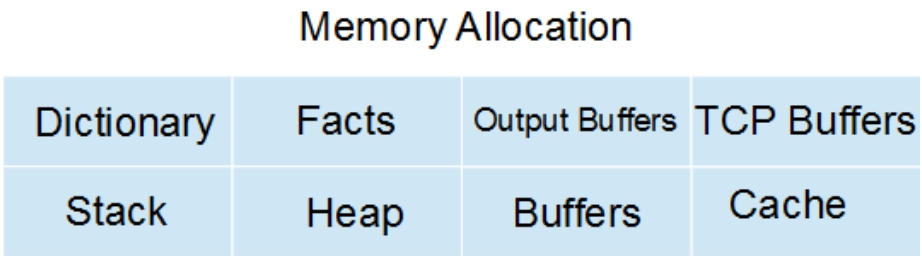

Figure 2: ChatScript data allocations

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, concepts, topics, user variables, even CS script. Strings are allocated out of either the stack (transiently for $_ transient variables) or the heap (for normal user variables and dictionary word names).

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.

```
typedef struct WORDENTRY //   a dictionary entry
{
    char*     word;
    union {
        char* userValue;
    }w;
    MEANING  meanings;
```

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.

A externally written fact looks like this:

```
( Symantec_Norton_Security kindof software x500000 8192 )
```

where the fact is contained within (). 3 main fields are shown, followed by a bit mask describing how to interpret the fields, and then a bot id identifying what bots are allowed to know this fact (this field is omitted if all bots can see it). Likewise if there the bit mask is 0, it too can be omitted.

Each field of a fact is either a `MEANING` (which in turn references a dictionary entry which points to a text string as name) 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 as above. Externally facts are stored as pure text way because user facts get relocated every volley and cannot safely be referred to directly by a fact reference number.

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 interpret the object field of the fact. JSON distinguishes primitive values (true, false, numbers, null) from strings, whereas CS uses strings for all of them and tracks how JSON writes them using bits on the fact. Here

```
( jo-225 bluetoothdevice iPhone x2200 32 )
( ja-2 0 DUMPDATA x1200 32 ).
```

The flags on these facts indicate that there are JSON facts (one a JSON object fact and one a JSON array fact). And that both have as their object value a JSON string. Unlike JSON which requires double quotes around all strings, CS requires no marker unless the string has internal blanks or double quotes. In which case, the field is written with backticks like:

```
( jo-225 bluetoothdevice `iPhone is good` x2200 32 )
```

so that we don't have to store fancy escape markers on various internal characters. The backtick is the character solely reserved for CS to use in a variety of ways, and any user input containing such will have theirs automatically convered to a regular quote mark. It applies to any fact field, JSON or not.

JSON objects and arrays are represented as synthesized dictionary entries with names like like `jo-5` or `ja-1025` for permanent JSON object or array and `ja-t7` for a transient array. The data for that array or object always has it's name as the first fact field. The numbers are generated uniquely on the fly when the JSON composite is created.

Each key-value pair of an object is a fact like `(jo-5 location Seattle x2200 )`. Each array element pair is a fact like `(ja-5 1 dog x1200)`. Array facts start with 0 as the verb and are always contiguously 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 JSON composite, like `jo-5` or `ja-t2`. That name, 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.

User Variables

User variables are names beginning with $ that can be found in the dictionary and have data (not their name) pointing to text string values. Their prefix describes who can see their content.

```
$varname - globally visible permanent variable in heap
$$varname - globally visible volley transient variable in heap
$_varname - locally visible transient variable on stack
```

Permanent variables are saved across volleys in the user's topic file. Local variables are only visible within the scope of locality (the current call to the topic or function).

Function Variables

```
outputmacro: ^myfunc(^var1)
```

Function variables like ^var1 are defined in the arguments list of a function (outputmacro/patternmacro) and managed by the system outside of the dictionary. They are like $_varnames (local transient) but should be rare and handle pass-by-reference data for the duration of execution of a function. They are only visible within the code of that function but they enable write-back onto the caller's variable passed in.

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.

Typically you assign into a factset via a query like:

```
@2 = ^query(direct_sv my animal ?)
```

Buffers

Input and output from CS involves potentially large text and so these have been preallocated from the heap as buffers. One allocates a buffer with AllocateBuffer() and one frees the most recent buffer with Freebuffer(). You don't have to pass a pointer to the buffer, you may only free them in the reverse order they were allocated. This follows the general memory management strategy of "onion layers".

Internal lists

The system needs to track lists of things, like global permanent variables whose values changed this volley (to write them out to user file). These lists are usually kept on the heap, though sometimes they reside on the stack. The list pushing routines are

```
HEAPREF AllocateHeapval(HEAPREF linkval, uint64 val1, uint64 val2, uint64 val3 = 0);
STACKREF AllocateStackval(STACKREF linkval, uint64 val1, uint64 val2 = 0, uint64 val3 = 0);
```

And the corresponding pop functions:

```
HEAPREF UnpackHeapval(HEAPREF linkval, uint64 & val1, uint64 & val2, uint64& val3 = discard)
STACKREF UnpackStackval(STACKREF linkval, uint64& val1, uint64& val2 = discard, uint64& val3
```

Memory Management

Many programs use malloc and free extensively upon demand. These 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 (a kind of onion layer), 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. It will be implicitly deallocated at the end of the volley. But during the volley one could imagine that some heap memory is no longer in use but hasn't been freed until the end of the volley. (This is something that garbage collectors handle but CS has only a manual garbage collection).

Fact memory is consumed by `CreateFact` or various JSON assignment statements and lasts for the duration of the volley.

But CS supports planning, which means backtracking, which means allocated heap memory which is nominally not pointed to anymore along the way is not "free" 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.

Onion Layers

When ChatScript starts up, it loads data in layers. You can free a layer at a time, but only the outermost one (hence the peeling the onion metaphor).

First layer 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 from the TOPIC folder. `Build0:` layer and then `Build1:` layer. 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 peeled off and the system is ready for another user.

All layers are "peelable". 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 peel a layer, 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.

Garbage collection

Using the layer model, normally user conversation causes heap allocations and dictionary allocations that can be rapidly and automatically released when the user layer is peeled. And plenty of space should exist to manage without any garbage collection occurring within the volley.

But if garbage collection is needed, scripts can invoke them. 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 script 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.

Nominally this means all facts and user variables you worked on after the mark will be lost. This is fine if your results are going to an external place (like writing to a file with ^log). Failing that, there are a couple of mechanisms for passing data back across a ^MemoryFree. You can write something onto a match variable. Match variables have their own memory and last until changed. Or when you call ^MemoryFree, you may pass either a single variable or a JSON structure on the call. Its value will be protected. The value is copied into stack space, the release occurs, and then the value is reallocated from the current full heap (now without the mark location). .

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 complex 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, or it's a word learned typically in first grade. Eg.,

```
typedef struct WORDENTRY //   a dictionary entry
{
    uint64  properties;
    uint64  systemFlags;
    unsigned int internalBits;
    unsigned int parseBits;
```

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 and layer 2 (boot layer). 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.

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 do facts.

```
typedef struct WORDENTRY //   a dictionary entry
{
    FACTOID subjectHead;
    FACTOID verbHead;
    FACTOID objectHead;

typedef struct FACT
{
    FACTOID_OR_MEANING subjectHead;
    FACTOID_OR_MEANING verbHead;
    FACTOID_OR_MEANING objectHead;
    FACTOID_OR_MEANING subjectNext;
    FACTOID_OR_MEANING verbNext;
    FACTOID_OR_MEANING objectNext;
```

So when you do a query like

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

CS will retrieve the list of facts that have `dog` as a subject and consider those whose verb is `love`. 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 ?)
```

work 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 time, local memory usage, 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 ($_var) or call by reference (^var). 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. Functions are potentially "owned" by a bot, 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. If the letter is uppercase, it means function supports variable argument count.

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.

All system functions return both a text buffer answer and an error code. For error codes see the section on error handling.

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.

```
* marks variable data coming preevaluated.
* marks ends of rules in scripts.
* quote values of variables and fact fields when writing out to the user's topic file.
* creates specific internal dictionary entries that cannot collide with normal words.
```

Script Execution

Execution, be it pattern matching or output processing, proceeds token by token. There is not really much of building up tokens on a stack and then popping them as needed (like calculator functionality). This means you cannot write parenthesized numeric expressions nor use function calls as arguments to other function calls.

Evaluation

Script execution involves one-by-one evalutation of tokens. It happens in patterns and if tests. It happens in the output side of a rule. It happens in function execution and it happens in interpreting the arguments to a function. It happens in active strings.

Numbers and words just evaluate to themselves. As do factset references like @4 and concept set names like ~animals.

User variables ($, $$, $_) evaluate to their value, unless they are dotted or subscripted, in which case they evaluate to their content and then perform a JSON object lookup. Or if the token immediately after it is an assignment operator of some kind (e.g. = += -= etc).

System variables (%) evaluate to their value. Not generally advised but for some system variables it may make sense to use them of the left hand of assignment statements.

Match variables (_0) evaluate to their content, original or canonical depending on whether the reference is quoted or not.

Function calls evaluate to their returned result and absorb tokens from (to) . Be advised you cannot usually use a function call as an argument to another function. You'll need to assign the first call to a variable, and then use the variable as argument to the second function.

Active strings ^"xxx" and ^'xxx' process their content in the moment of use.

Spacing

The script compiler normally forces separate things into separate tokens to allow fast uniform handling. E.g., `^call(bob hello)` becomes `^call ( bob hello )`. This allows the `ReadCompiledWord` function to grab tokens more easily.

Prefix characters

The engine is heavily dependent upon the prefix character of a script token to tell the system how to process script. The pattern prefixes are:

```
=   to designate a comparison operator
!   to invert a Test
?   a unary existence operator (prefixing a variable)
@_1+ to indicate jumping in a pattern to a location and altering direction (+ or -)
```

Prefixes also used in patterns and output code are:

```
$   to indicate user variables
%   to indicate system variables
_1  to indicate a specific match variable
^   to indicate function calls, function in-out variables
~   to indicate a concept set (but this has no impact on execution of tokens)
@1  to indiate a specific numbered factset
'   to indicate to use the original value of a match variable or word
#   to indicate a system defined numeric constant (or to indicate comment to end of line)
^^  to indicate indirection assignment thru an in-out function variable
```

Skip codes

The script compiler puts in characters indicating how far something extends to allow faster execution. This jump value is used for things like if statements to skip over failing segments of the if. It is used for knowing where a pattern label ends and even for skipping over entire rules.

Skip codes (1 or 3 characters that represent a number) allow the system to move rapidly in the code stream.

Rule and Label skips

Every rule starts with a skip code that tells the offset needed to skip past this rule and move to the next rule. The system view of a rule is normally starting with the kind (u:) but it can back up 4 characters to read the skip code if it needs it. There is also a 1 character skip code in front of any rule label, that tells the width of the label to get to the opening paren of the rule's pattern.

```
00J u: <RULE_LABEL ( test ) This is a rule.
```

Comparison skip

Comparison conditions in patterns use a 1 character skip at the start of a comparison token to index to the start of the operator (i.e., to know where the end of the variable name is). The leading `=` is the prefix code that says this token is a comparison token.

```
u: ( =6$foo==5 )
```

IF and LOOP skips

`if` and `loop` are actually functions. And they use 3 character skip codes to move around the pieces.

```
u:  (test) if ($val) {do this} else if ($val1) {do that} else {anyway} .
u: ( test ) ^if 00c( $val ) 00j{ do this } 00? else 00d( $val1 ) 00j{ do that } 00y else ( 1
```

The second line shows the compiled form of a complex `if`. `if` is compiled into the ^prefixed form (since it is not something to echo as a word to the user). The 00c skip code before the first test is the offset to get the other end of the first test, where another skip code lies. Whether the test succeeds or fails, this will get us past the test immediately. The next skip code (00j) can take us past the code to for this branch of the if. If the test fails, you would use that skip to bypass the "do this" and point to the next `else`. The skip code after the end of the "do this" process tells how to skip to the end of the entire if, so normal execution can resume.

Loops uses a skip code, to allow it to jump to the end of the loop when the loop condition fails.

```
u:  (test)  loop(1) { do this }
u: ( test ) ^loop ( 1 ) 00g { do this }
```

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))`. One can even generate a fake sort of concept set, by doing `^createfact(dog member`

`pet))`. This causes references to `dog` to automatically mark `pet` and propogate along any concepts of including that.

In the TOPIC folder are 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.

Pattern matching

Patterns can exist at the start of any rule type. And they can exist inside of `IF (PATTERN ...)` statements.

Token processing

Pattern matching executes the pattern stream token by token. A token might be a word, or a wildcard, or an entire assignment statment with active string

like '$tmp:=^"This ^compute(a sub b)". The script compiler will force spaces between tokens, particular ones with special priority like < or [or (.

```
u: (<[a b c])  -- sample script
u: ( < [ a b c ] ) -- compiled notation
```

The exception to the token by token rule is function calls, where when the function name token is detected, it gets control over the pattern stream to manage the `(` and arguments and `)` .

Legal gaps

A top-level pattern `( a )` implicitly means `( < * a )` except that since we dont actually use the `<`, backtracking will work. When a substantive (non-wildcard) match happens, the system moves the current marker of where we are in the sentence to that position. However, the movement from the prior position to this new one must be legal. When wildcards are involved as the predecessor token, they define how much gap is allowed. when non-wildcard tokens are involved, there is no gap and the new match must immediately follow OR be coincident within the current position.

Backtracking

Unlike regex or prolog, CS pattern matching does not do full backtracking on failure (as that would be expensive). Instead if the system tracks the first real word it matches, and if the pattern fails, it can retry the entire pattern starting after that first match. Anything that forces a location in the pattern will inhibit this, e.g.,

```
u: (< test)
u: (@_0+ try)
```

The above patterns, when reexecuted, would force the start of matching again, so backtracking is not feasible.

Because backtracking is limited, one should prefer using concept sets to using `[ xxx yyy zzz]` because concept sets find the closest match to your current position whereas the [ ] try to match in order, and thus may find matches much later in the sentence. Additionally one should strive to try to find the most significant words first, ones less likely to be found in the sentence. E.g.,

```
u: ( I want * the adjustment )  -- can find the anywhere and never backtrack to adjustment
u: ( I want * _adjustment @_0- the)  -- finds the rare word adjustment and then confirms the
```

Pattern macros

Pattern macros are merely patterns encased by a function call. In execution, when a pattern macro is encounterd, the system temporarily changes its pattern stream pointer to use the definition of the macro and then on failure or match, restores the old stream to continue. This only works for 1 call; you cannot nest a call to a pattern macro from within a pattern macro.

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.

Error handling

There are two kinds of error handling, rule-based and engine based.

All script-callable engine Functions are declared `FunctionResult` and return NOPROBLEM_BIT when things are fine. Other codes indicate whether to terminate a rule, topic, sentence, or input as OK or an error. In script you can suppress errors from below by kind, e.g., `^nofail(TOPIC ...)`. Or you can trap all errors and examine what termination happened via `^result( ... )`.

The engine tries to detect and recover from various error conditions. It uses `ReportBug` to add an entry into LOGS/bugs.txt (and the server log if a server). Any error message starting with `FATAL` means that the call will not return; CS will end execution instead. Fatal errors are not trapped by a try/catch or setjmp/longjump mechanism because it is too risky. Bits in the dictionary may have been changed or some kind of memory overwrite may have happened. This would leave the server damaged for all future volleys. Instead, the server exists. External processes restart the server (and may retry the volley) and the fully reloaded server will be clean and ready to continue.

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. And ^flushoutput.

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 of 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 into 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.

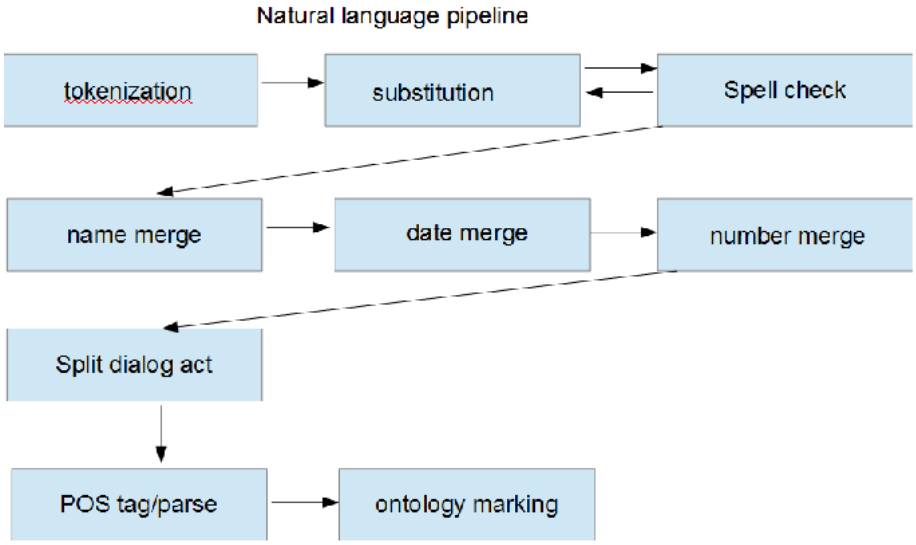

Figure 3: ChatScript NLP pipeline

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*, *bat*, *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.

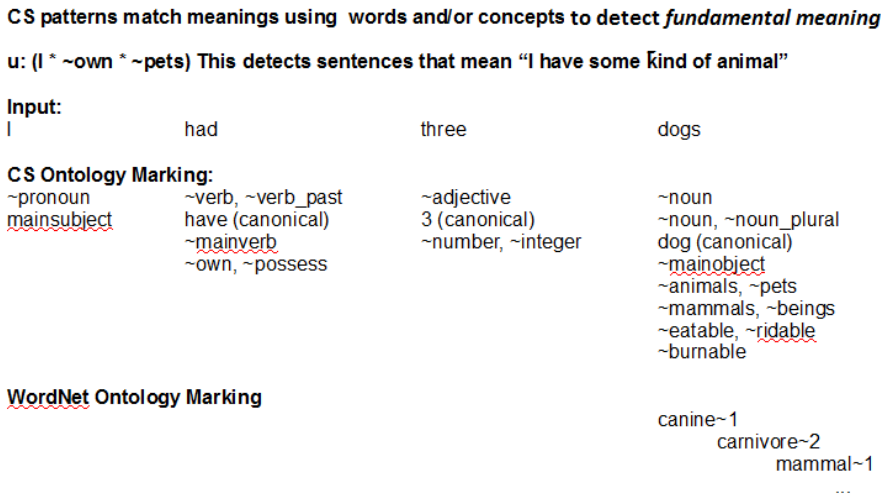

Figure 4: ChatScript Ontology

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.

Multiple Bots

ChatScript can run multiple bots in one build. Each bot can own (or share with specific other bots) topics, facts, and functions. This means multiple instances of these things may exist, owned by different bots. A bot in a multi-bot environment has a unique name and unique id. The botmacro defines the name and within it the bot id is declared by assignment to $cs_botid. You define in the filesxxx.txt file what bot name and id is compiling some particular piece of script.

```
outputmacro: mybot()
$cs_botid = 1
```

Bot id's are powers of 2 and you are limited to 64 unique bots.

You define who owns a topic using a topic flag or in a bot zone of your filesx.txt. This field is saved with the topic data, with spaces around the bot name so one can do a `strstr` on that field to see if current bot name is found.

You define who owns a fact normally in the bot zone of your filesx.txt. A fact field has a bot id mask saying which bots may view it. A mask of 0 means all bots can see it. Code that accesses facts uses functions like GetNondeadHead and GetNondeadNext, which walk a fact list ignoring all facts marked as killed or not visible to the current bot (myBot).

You define who owns a function just like a fact and the function data includes the mask. The only caveat is that any botmacro functions must have a 0 value (must be globally visible) in order to allow new users to be launched. Code is stored in TOPIC/BUILDn/macros1.txt (if defined in BUILD1). A sample definition:

```
^oob o 1 0 B( $_what $_x $_y) $_x = 1 $_y = $_x `
```

The function name is ^oob.

The `o` says this is an output macro. `t` is a tablemacro. `d` is a dual macro and `p` is a pattern macro. 'O' and 'P' mean that the function allows variable numbers of arguments up to its actual limit count. Arguments not supplied will be defaulted to null.

`1` is the bot id owning this function.

`0` are any flags on the arguments that specify special processing of incoming values.

`B` is the argument count as the number added to 'A' (i.e. 1 argument). If arguments exceed 15, the base will be 'a' instead with 15 deducted from the index.

The list in parens are the function arguments and any local variables that need to be saved and restored across the call. Following that is the actual compiled script code, ending with a backtick which completes this definition.

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"},
```

Documentation

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