ChatScript Engine Manual

Revision 10/23/2014 cs6.86

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.

Data Structures & Data types

Part of understanding CS engine programming is understanding the basic data available to CS. The fundamental datatype is the text string. Text strings represent words, phrases, things to say. They even represent numbers (converted on the fly when needed to float and int64 values for computation). Text strings represent the names of functions and concepts and topics. Some of these text strings have attributes attached to them and are kept in a dictionary for easy lookup. The other fundamental datatype is the fact, a triple of data. Each field of a fact can be a text string (stored in the dictionary) or a direct reference to another fact. Facts represent relationships among text strings. When stored in variables, a fact reference is a number, which in turn is a text string. User variables hold text strings. Match variables hold text strings, but a match variable is more complex because usually it holds data found in an input sentence. As such a match variable holds the original value seen in the sentence, a 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.

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, 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 phrases like “read my lips” in a pattern or keyword list are treated as words in the dictionary.

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. Even things that are not words, including phrases, can reside in the dictionary and have properties, even if the property is merely "this is a keyword of some pattern or concept somewhere". 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 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. 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 using facts (using the "is" verb). Word are conceptually linked 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.

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.

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. It allocates everything in advance and then (with rare exception) it never dynamically allocates memory again, so it cannot fail by calling the OS for memory. And 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. And incorporated extensions, like Curl, JavaScript, PostGres, Mongo, etc all do their own memory management that CS cannot control.

Memory is divided into dictionary, fact, string space, and buffers. All text (which is not being computed on the fly for various actions in script) goes into string space. That which is loaded as part of layers 0 and 1 are considered permanent.

That is which is generated during the user volley is considered "transient" and will be released at the end of the volley. String space itself uses AllocateString to allocate memory for the duration of the volley and AllocateInverseString (which runs in the opposite direction in the same string chunk) to allocate memory that is only supposed to last for the duraction of a function. And there is AllocateBuffer, which has a limited preallocated collection of 80K buffers (current output size limit) that can be allocated (FreeBuffer) to release within some scope.

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.

The other problem is that memory is not released until the volley is over. So conceivably memory is free but hasn't been freed. 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 a memory mark and memory free function so you can explicitly control this while reading a document.

Run-time Model

The fundamental units of computation in ChatScript are functions (system functions and user outputmacros) and topics.

System functions are predefined C code to perform some activity most of which take arguments that are evaluated in advance (but some wait until they get them to decide whether to evaluate or not).

Outputmacros are scriper-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, but these arguments are typically not evaluated. This is not pass by value. Outputmacro code is executed as though it were directly spliced into the original caller's code. ^args are processed by converting them into what the caller was using, except for format strings ^"xxx" which are evaluated in the caller's context before being passed as an argument to a macro.

Calls to engine functions may evaluate their arguments or pass the raw script stream, depending on what they need to do. Calls to user-defined outputmacros will normally just pass their arguments untouched (pass by reference) so the macro could write back onto the variable passed. The exception to this is passing a local variable like `$_xx`. Since nobody outside the caller is allowed to access or

touch this variable, the system computes the value and passes that, along with a “ marker prefixing it. That way various system routines can detect that it is already evaluated, not reevaluate it, and not try to write back onto it.

Script Execution

The scripting language is heavily dependent upon the prefix character to tell the system how to behave. 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.

Pos-parsing

The system runs pos-parsing in two passes. The first pass is execution of rule from LIVEDATA/SYSTEM/ENGLISH 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 anomolies.

Queries

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 executed to perform the query.

The Dictionary

The dictionary consists of WORD entries, stored in hash buckets when the system starts up. Once system startup is complete, those buckets are closed and new entries created from user input are all stored in bucket 0, where they can be undo easily. Linear search within this bucket is unimportant because the user

won't create many new entries. So when the dictionary performs word lookup, it uses the hash table first, and if it finds nothing there, it uses bucket 0. The hash code is the same for lower and upper case words, but upper case adds 1 to the bucket is stores in. This means all forms of upper case of a word hash the same, so there is only 1 actual print form of an upper case word available.

Marking

The system takes the input and splits it into the original input and a canonical one. Both are "marked". 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. 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.

Spell Checking

Spell checking takes a word it doesn't recognize and performs a variety of attempts. These in include merging it with adjacent words, splitting it into two words, adding/removing hyphens, or hunting among words whose length is plus or minus one letter, to try making a minimal edit distance.

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., = before a comparison token or * 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 VARIABLE_ARGUMENT_COUNT means args evaled but you have to detect the end – ARGUMENT(n) will be ? STREAM_ARG – 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"},