ChatScript Overview of Input to Output

While there are many natural language tools and engines out there, ChatScript is unlike most (or all) of them. Academic systems are built of independent components that are trained on Wall Street Journal perfect capitalization and grammar sentences. Chat isn't that. The user will often type in all lower case or all upper case and omit punctuation of all sorts and use texting shortcuts. ChatScript is aimed to survive that. While ChatScript will do pos-tagging and parsing, in general a normal chatbot does not use that information in its rules since chat input isn't reliable. Instead rules will typically be based on patterns of sets of keywords to hunt for specific meanings.

So how does Rose go from the input *is seattle in california* to the output *No, Seattle is in Washington.*

First, the incoming volley has a user associated with it, so ChatScript loads the user's specific state file; that tells it what topic it was last in, when the last volley was, what's been said, what rules have been used up, what private facts have been generated, etc. All of the dictionary and scripts themselves as well as world data facts are always permanently loaded and so we are just adding a small user veneer into the system.

ChatScript is going to run multiple passes of script topics on every volley. A volley is what the user inputs (can be multiple sentences) through to what ChatScript outputs (can be multiple sentences).

There is `$control_pre`, which is run before any user sentence is analyzed. This allows you to initialize or clear your environment variables or whatever. `$control_main` is run on each user sentence. `$control_post` is run after all user input has been processed, and allows you to examine all the outputs generated to do postprocessing like generating additional emotional data or detecting pronoun resolution information.

ChatScript starts by transforming input words using substitutions files. It has files for texting, British spellings, common spelling mistakes, contractions, abbreviations, noise (like *hmm*), and interjections mapped as speech acts (sure = `~yes`, how are you = `~howzit`, good night sleep tight = `~emogoodbye`). These are all in `LIVEDATA`, meaning they are not baked into the dictionary but loaded on startup. Trailing punctuation is removed, with bits set to reflect punctuation status (question, statement, exclamation).

Sentences beginning with an interjection/dialog-act are split off into their own sentence, so *yes. I think so.* and *yes, I think so* are the same two sentences. Proper names are then merged into single tokens (the "named-entity extraction concept) so the 4 tokens *I love John Hardy* becomes 3 tokens, *I love John_Hardy.* Similarly word numbers are merged so *I ate two million and twenty four birds*

becomes *I ate two-million-twenty-four birds.* The sentence is then spell-checked and revised from that if needed.

This smoothed input is then passed to code that generates all the potential part of speech things each token might be (where parts of speech include crossover roles like a noun acting like an adjective, a verb acting as a noun, etc). At this point the input is actually split into two streams of tokens, the original and the canonical. *My dogs are fun* is an original stream (with pos markings) and *I dog be fun* is a canonical stream.

A rule-based pos tagging pass analyzes the stream from either viewpoint and tries to eliminate possible pos meanings, without being wrong and eliminating a valid pos tag. This is done without trying to understand the structure of the sentence- it is unaware of prepositional phrases, what the main subject is, etc. The next phase runs a parser that is aware of structural needs and combines deciding on structure with forcing additional pos-tag restrictions as it goes. It is designed to be a "garden path" algorithm, so it walks the words in order, looks ahead whatever it can, and guesses what is happening. If it later discovers an inconsistency it will try to look back and revise a decision. But the algorithm is not interested in generating all possible legal parses, just a most likely possible parse.

With the pos-tag/parse in hand, the system then decides the tense and whether the sentence is a question even if it lacks a question mark.

ChatScript then marks all words in the sentence with what concepts they belong to. *My dogs are fun* will have markings on dogs that reflect its part of speech, its role in the sentence, the word dog, dogs, and concepts like `~pets`, `~animals`, `~beings`, etc. Not just individual words are marked but series of words are marked as well, so the system can mark a movie title or an idiomatic phrase.

Finally the system is ready to run script on the input. Except, the first script it will run is not your stuff that generates output. Instead if you have one, it will run another preparatory script to revise its analysis. You can decide that something should be considered a question (e.g., I treat *Tell me your name* as a kind of question), you can revise the inputs (I treat all hypothetical sentences like *if you live forever, will you die* as two sentences *assume you live forever. Will you die*), you can revise inputs substituting pronoun values for pronouns, set additional marks on idioms (I mark *what do you do* with the `~work` topic).

Now finally we get to your normal main control script. Bear in mind that all of the above analysis phase is controllable from script (you don't have to have spell correction). Also bear in mind that you can redo the analysis under different constraints at any time. For example, I have spelling correction on, but if my script detects that the user is offering their name, I turn it off and tell input to reanalyze. The odds of them spelling their name wrong are low and the odds they have an unusual name are high, so an input like *My name is Haro Varis* will be managed by changing the conditions of analysis after the fact.

The control script will directly invoke some topics unrelated to the content of the sentence. Other topics will get called because there are words in the sentence that have appropriate keywords for the topic. In the case of "is seattle in califoria", the markings of `~city` on Seattle and `~state` on California will route it to my geography topic. Where a pattern like:

```
?: ( << is _~city in _~state >> ^query(direct_sv _0 part ? 1 ))
```

will catch it. Note the `^query` call. It could be inside the pattern or outside at the start of the output. Makes no difference really. It goes and looks up geography facts that ship with ChatScript to see if any such facts exist. Such a table fact is:

```
(Seattle part Washington)
```

Nothing says the fact is the one we want, merely that we know data about Seattle's relationship to the world. Had the question been *is seattle in europe* we'd still be looking at the same fact, which while not useful for the question yet, tells us we know the relationship of Seattle in the world. The actual output script will have to determine what fact is really needed, by chasing around the facts relationships. Eventually it will determine if Seattle is in the named state or continent or planet or solar system or whatever, and make its planned response.

Grist for the mill of scripts is the fact that ChatScript supports introspection. You can see into its workings. You can find out what kinds of input substitutions were performed. You can change marks and tense and punctuation on the fly or analyze an arbitrary sentence. You can alter the availability of rules and topics, you can retrieve rule data itself, which means you can write scripts to replace how ChatScript's various routines normally handle things. For example, I have my own `^rejoinder()` handler to allow the system to manage yes/no inputs which have follow ups. This allows the system to see that *yes I love chicken* can be handled by ignoring the yes handler and moving directly to I love chicken under appropriate circumstances, while *yes. Do you go to football games* will probably be accepting the yes rejoinder branch and treating the next input as unrelated. The postprocessing topic is all about introspection- seeing what the chatbot generated for output, what rules and topic it came from, etc.

With output generated, the system returns to any next input sentence. When all sentences are handled, the system invokes the post processing script. Then it outputs whatever it has to the user, writes out the user's status file and resets itself back to a clean state, ready for new input from an arbitrary new user.