ChatScript Practicum: Concepts and Meaning

Revision 4/26/2019 cs9.3

‘”There’s more than one way to skin a cat”’. A problem often has more than one solution. This is certainly true with ChatScript. The purpose of the Practicum series is to show you how to think about features of ChatScript and what guidelines to follow in designing and coding your bot.

Lewis Carroll got it right back in 1872 in “Through the Looking-Glass”

```
"I don't know what you mean by 'glory", Alice said.
Humpty Dumpty smiled contemptuously.
    "Of course you don't-till I tell you.
    I meant 'there's a nice knock-down argument for you!' "
"But 'glory' doesn't mean 'a nice knock-down argument'," Alice objected.
"When I use a word," Humpty Dumpty said, in rather a scornful tone,
    "it means just what I choose it to mean-neither more nor less."
"The question is," said Alice,
    "whether you can make words mean so many different things."
"The question is," said Humpty Dumpty,
    "which is to be master-that's all."
```

Natural Language is all about getting computers to understand meaning from what humans say. But meaning is a slippery beast because there is no fixed meaning. Meaning as expressed in words requires agreements between the speakers about what the words represent. Two speakers often don’t fully agree on meaning because even if they agree what words represent, they don’t have the same context. Interpreting meaning involves being able to build intuitions of what will or did happen and all sorts of context not immediately there in the words. Some of that even comes from one’s personality and personal life experiences, which are not necessarily shared by others. A simple phrase “I hate it” can be recognized as dislike, but could range from a passionate loathing to a simple don’t really care for it. There will be entirely different implications coming out of that.

In English (and many other languages) words can have multiple meanings.

```
He swung the bat around and around before it flew away.
```

If we are at a baseball game, we have a context for a wooden bat object, though this sentence needs prior ones to establish what `he` stands for. And a knowledge of physics and common behaviors to know that swinging something around and around and then letting go will have it flying away but not of its own volition. And that it will not travel far before striking the ground or some object.

If we are in a zoo and watching a zookeeper, then we have a context for a bat animal and a knowledge that likely the bat flew away of its own accord, maybe

because it didn't like being swung around. And we might have expectations that the bat will keep flying until it is out of sight.

Concepts

ChatScript does not `understand` anything on its own. The engine does not have meaning built in. Developers write rules to hunt for specific meanings. The bot can only `understand` the user if they say something the bot has been taught to look for. Of course the bot can have rules that can detect meaning from the very precise ("I eat dogs for breakfast on Tuesdays on my patio") to the mere detection that the user asked a question or made a statement. A bot can fake understanding to everything said to it if it merely says "I agree" when someone makes a statement or "I don't know" when someone asks a question.

To write these rules we have patterns. And to give our patterns generality, CS has concepts. Actually, there are horizontal concepts and vertical concepts.

Horizontal concepts

Horizontal concepts are defined using 'concept: ~name (value1 value2 ...). Concepts are lists of words that share some common property. We abstract a specific possible meaning from a word by including it in a concept list. Then we use that concept in patterns.

```
concept: ~hungry (ravenous hungry starving)
concept:  ~animals (bat cat platypus)
concept: ~baseball_items (bat ball base)
```

Sometimes concepts represent actual synonyms (~hungry). Any of the words in the concept can be used in all the same sentences, leading to patterns like:

```
( I * ~like * ~food)
```

which allows us to detect the meaning of positive user taste preferences in some food with a single rule. The cool thing about concepts is that a concept can have a single member or thousands of members and the processing of the pattern runs in constant time regardless. Synonyms are handy in that if we agree on the meaning of one word, we can interpret synonyms to all have the same meaning.

Sometimes concepts represent affiliated items that are only equally usable in a sentence under specific attempts to understand meaning. If you want to know if someone is talking about a living being, you can use ~animals to allow any of its members in sentences. If you want to detect that someone is talking about the game of baseball, without gathering any additional meaning from the user's input, then the ~baseball_items can be used in a pattern.

Note there are two ways to define a horizontal concept. One is the concept declaration.

```
concept: ~conceptname( word1 word2 ...)
```

Or you can use ^createfact (like in a table) to create concept members. The concept declaration is merely shorthand for creating all the relevant facts. Commonly tables are used to simultaneously define members of one or more concepts and to create relationships between words in the table. Like a table with lines of car brands and car models, such that you can know a word is a car model name and also know what car company makes it.

Even so, it is best (but not required) to declare the concept as a stub somewhere. because then CS can warn you if you already have a concept of this name. You can create those facts before and/or after you have defined this stub. I typically write the stub and then the table code that creates the member facts of it.

```
concept: ~carmake() -- stub
concept: ~carmodel() -- stub
table: ~cardata($_carmake $_carmodel)
    ^createfact($_carmodel member ~carmodel) -- adds a member
    ^createfact($_carmake member ~carmake) -- adds a member
    ^createfact($_carmodel builtby ~carmake) -- adds a relationship
DATA:
Ford Ranger
Nissan Sentra
```

Vertical concepts

A word in the dictionary can have multiple meanings. You already know that and you can see this in ChatScript by using the debug function `:word`.

```
:word eat
  Meanings:
    verb
    6: eat~6 worry or cause anxiety in a persistent way
      synonyms:  eat_on~1  *eat~6
    5: feed~5 take in food
      synonyms:  *feed~5  eat~5
    4: eat~4 take in solid food
    3: eat~3 eat a meal
    2: eat~2 use up
      synonyms:  run_through~1  deplete~1  use_up~1  eat_up~1  consume~3  wipe_out~5
    1: eat~1 cause to deteriorate due to the action of water, air, or an acid
      synonyms:  rust~9  corrode~2  *eat~1
```

The above call to `:word eat` dumps dictionary content about it. We can see it is only considered a verb (has only verb meanings) and there are 6 different

meanings, each with its own synonyms. You can refer to a specific meaning in a pattern or concept by using the ~n suffix. E.g., `eat~3` means `eat a meal`. The worst word in the dictionary is `break` with 63 noun and verb meanings.

Along with the above data, the CS dictionary (based on WordNet from Princeton University) defines vertical concepts because the words are linked into an ontology. An ontology is a hierarchy of words (particularly nouns) such that a word lower in the hierarchy inherits all the characteristics of its upper parents. This is an "is" fact relationship (concepts use "member" facts). A canine is a dog. A dog is a mammal. A mammal is an animal. And so on. Things which are true of the higher level are also true of the lower representations of it. If an animal eats food, then by definition a canine does too (assuming still alive).

```
:up canine
  canine~1: (fissiped mammal with nonretractile claws and typically long muzzles)
    is carnivore~2  (a terrestrial or aquatic flesh-eating mammal)
       is placental_mammal~1 (mammals having a placenta)
          is mammal~1 (warm-blooded vertebrate having skin covered with hair)
             is vertebrate~1 (has bony/cartilaginous skeleton and segmented spine)
                is chordate~1 (of the phylum Chordata having a notochord or spine)
                   is animal~1 (living organism characterized by voluntary movement)
                      is being~1 (living thing with ability to function independently)
                         is animate_thing~1 (a living entity)
                            is whole~1 (an assemblage of parts regarded as a single entity)
  canine~2:    (1 of the 4 pointed teeth between incisors and premolars)
    is tooth~5  (hard bonelike structures in the jaws of vertebrates)
       is bone~2 (rigid connective tissue making up the skeleton of vertebrates)
          is connective_tissue~1 (tissue of mesodermal origin)
             is animal_tissue~1 (the tissue in the bodies of animals)
                is tissue~1 (an aggregate of cells of similar structure and function)
                   is body_part~1 (any part of an organism such as an organ or extremity)
                      is piece~11 (a portion of a natural object)
                         is thing~1 (a separate and self-contained entity)
                            is physical_entity~1 (an entity with physical existence)
```

The :up command starts at the lowest point of the word and shows its higher `is` relationships. It's confusing to say up and have the printout display downwards on the page, I admit. Just remember that more indented means higher up in the hierarchy. Using the `:up` command one can see that one meaning of canine is as part of the body and the other is as a dog. If you use a named dictionary meaning in a pattern like `(canine~1)` it means that any of the meanings below it in the hierarchy (above it in the text display) also match. So if a pattern is `(tissue~1)` then that will match an input of `canine`.

Pseudo concepts

You can write patterns with what looks like a local concept, the [] or {}.

```
concept: ~actors (~animals ~family_members spider)
u: ( [ ~actors (tasty bugs)] * ~like *1)
u: ( {~actors (tasty bugs)} * ~like *1)
```

The [] requires one from the set of given values whereas the {} optionally allows one from the set. This is different in two ways from a true concept.

First, most obviously, you can use patterns inside the pseudo concept like `(tasty bugs)`. You can't define that in real concepts, though you can create the effect of that by making a rule that marks a concept based on the pattern.

```
u: ( _(tasty bugs)) ^mark(~actors _0)
```

Hereafter, patterns can use ~actors and get the results of the `(tasty bugs)` match also.

Second, a true concept matches in linear time (unrelated to how many members it has) and matches the nearest match it can find. These pseudo concepts march down the elements one by one testing for a match. This means the time to match depends on the number of elements. It also means that it doesn't find the closest match to the set, it finds the first match of the set.

```
u: ([I we child baby])
```

Given an input of "The child is my father", that pattern matches "my". You have to be careful of canonical values around pronouns because maybe what you really wanted was `'I`to be matched. Still, if you changed the pattern to `'I` and if your input was "The child is mine because I raised it", again you will still match 'I' from later in the sentence and skip past `child`. It behooves you to put the least frequently used words first in your list, to improve the odds of a valid match. Or replace those words with a concept set actual reference, particularly if your pseudo concept is used in multiple rules. In particular if you use a concept instead of a pseudo concept, you only have to edit one place when you want to alter it. If you did it in each pattern, you have to remember to edit all of them, and likely they will start getting out of sync with each other.

Multiple meanings

None of this directly solves the problem of multiple meanings of words. Some words like `geologist` only have a single meaning in the dictionary. But lots of words have multiple meanings. And any word can be included in multiple concept sets (horizontal and vertical). What ChatScript does is make available all meanings of a word at a place in the sentence as `marks`. And since concepts can be included as elements of other concepts, this means CS will follow out all

encompassing concepts around a word. CS marks available dictionary definitions and all concepts it is involved in. Marking words happens once, at the start of sentence processing. Thereafter patterns can quickly look for the marks that match the word or concept referred to by a pattern.

One can use the debug command `:prepare` on a word or sentence to see all the detected markings.

```
:prepare eat
1: eat (raw):
   +eat
       +~eat
         +~swallow_food_verbs
           +~swallowing_verbs
             +~ingesting_verbs
               +~use_fooddrink
               +~bodily_function_verbs
                 +~body_verbs
                   +~animate_verbs
                     +~verblist
                   +~active_verbs
                     +~use_intentionverbs
                 +~usefulfactverb
     +eat~1
     +eat~2
     +eat~3
     +eat~4
     +feed~5
     +eat~6
1: eat (canonical):  //
```

The `:prepare` debug command will tell you all the things it has marked both for the original word and its canonical form. Again indentation is showing hierarchy relationships, since concepts can be nested inside concepts. You can see :prepare marked various concepts as well as all the dictionary meanings of eat.

Note that `eat~5` is not listed as marked but that `feed~5` is. That is actually a synonym meaning, which is used internally as the canonical value of that meaning when following up the hierarchy. You don't have to pay attention to it. A pattern `( eat~5 )` will react to both `eat` and `feed` as inputs.

So.... given that CS marks all possible meanings, how can we reduce the possibilities?

Restricting meaning

Restriction by part-of-speech

Sometimes ChatScript can restrict the possible meanings of a word if it can correctly determine the word's type (noun, verb, adjective, etc). Then if you have included the words type when you define the concept, you will reduce the set of possible meanings. You type a word by suffixing it with ~v for verb, ~n for noun, ~a for adjective/adverb.

```
concept: ~harmful_action (break~v cut~v)
concept: ~injuries (break~n cut~n )
```

For the above concepts, an input of "I cut myself" will mark ~harmful action on cut but not mark ~injuries, because `cut` was used as a verb. And an input of "I have a cut" will mark ~injuries and not ~harmful_action.

This ability depends on accurate tagging (detecting and marking) of parts of speech and even the best taggers are only about 97% accurate. So by default CS does not "trust" its tagging and will ignore your part of speech suffix. The assumption is that false detections are better than missed detections, because you can write script to fix false ones. If you want to have CS trust its pos-tagging, then you need to launch CS with the parameter `trustpos`. The engine will now obey the ~n and ~v kinds of suffixes to words in patterns and concepts.

Restriction by topic

The simplest way to restrict meaning is to know the context you are in. If all your TV rules are in a ~tv topic, then once you are conversing in that topic, those rules have first priority at interpreting the meaning of the user's input. Assuming we have a concept listing TV shows and `Lie to me` is on that list, then the input `I like lie to me` will have as one detected meaning `~tv_show` for the sequence of 3 words, in addition to all the individual meanings. And a pattern hunting for TV shows there will react.

Restriction by ^unmark

You can write rules to decide whether or not to kill off certain meanings (or even add more). TV show names can be anything, after all. So maybe you should turn off that meaning if there is no context supporting it. Here is such a rule:

```
u: ( << _~tv_show ![$$currenttopic==TV watch view channel television TV ] >>)
    ^unmark(~tv_show _0)
```

This rule wants any TV show name to be in some obvious context of a TV. If not, then the concept reference is removed and later rules will not be aware there

is a possible TV show reference there. Sometimes I have a preprocess topic that does this. Sometimes I do this only in topics where it specifically matters. In a category about cars, when detecting car models, a camper normally means an RV. But Volkswagon has a model called Camper. So to distinguish...

```
u: (_camper)
    ^refine() ^retry(RULE)
    a: (@_0- [Volkswagen VW] )
    a: () ^unmark(~carmodels _0)
```

The above rule will allow any camper preceeded by the car maker but remove all other campers so they don't mean car model. That's the form where by default we kill almost all references to camper. The other way is to allow almost all references. For car model `Dodge Charger`, we can be optimistic and remove only some charger references e.g.,

```
u: ( _charger)
    ^refine() ^retry(RULE)
    a: (@_0+ { into for} {my a} [hooked battery cellphone phone  ])
        ^unmark(~carmodels _0)
```

With the charger rule, we don't allow `charger for my phone` to detect ~carmodel anymore, but we can allow many other references that might be the car model.

While you can do this without using ^unmark in a specific pattern to detect the car model, if you have multiple rules that need to try matching then you don't want to repeat these patterns for each rule, and therefore the ^unmark trick is better.

Note that you are not limited to unmarking concepts. You can unmark words and even specific word meanings. Unmarking a word does not remove it from the sentence, only from detection.

Question or statement

Just as CS labels words with concepts, it labels sentences with punctuation, which act like simple concepts or words that you refer to in patterns to help detect meaning. A sentence is either a question or a statement. A statement can be a normal statement or a command (implied subject is "you") or an exclamation.

```
u: (? test)  --- this must be a question
u: (!? test) --- this must be a statement
u: (\! test) --- this statement ends in exclamation mark
```

A question is based on the user having a ? or the sentence structure looking like a question. A statement is anything not a question. A command requires an

initial infinitive verb. And an exclamation requires an exclamation mark.

CS can get it wrong and you can change those markings. For example, a construction like "should I get it" is a question. But that can be trumped by this rule:

```
#! Under no circumstances should I get it
u: (< under no [condition circumstance] ~auxverblist )
    RemoveTokenFlags(#questionmark)
```

English grammar considers "what do you know about" to be a question and "tell me about" to be a command (imperative statement). But my bots typically want to consider both to be questions. Or I want to consider speculation to be a question. So I have rules like these:

```
#! I wonder if
u: (< 'I {be} wonder )
    SetTokenFlags(#QUESTIONMARK)
#! please Tell me about birds.
u: (!'I < *~1 [~describe ~list ~explain  tell define ] %tokenflags&#COMMANDMARK)
    SetTokenFlags(#QUESTIONMARK)
```

^replaceword

An input sentence is a series of words. They get annotated with marks. And after that, what happens in pattern matching is all about the marks. CS will have marked not only concepts and such, but the actual words. And you can mark and unmark these things to your heart's content. The actual words in the sentence are ignored in matching a pattern. But they need not be neglected. You can change them without changing pattern matches using ^replaceword.

```
u: (_elephant)
    ^replaceword(giraffe _0)
u: (_elephant)
    _0
```

This does NOT automatically mark giraffe here and all its concepts. It just changes the word. So the second rule detects the elephant but prints out `giraffe`. (You can create a full replacement effect by ^unmark on the elephant and related concepts and ^mark on giraffe and related concepts.)

So aside from being a curiosity, what is the practical value of this ability?

Consider this input sentence "my doc gave me pills". The word `doc` can be short for `document` or `doctor`. You can write script to decide which it is and alter the marks around it. But the pattern memorize operator `_` grabs the actual word of the sentence. So if you try to use what is memorized it will be the word `doc`. But if you use ^replaceword to change it to `doctor`, then in later rules that

will be what gets memorized, not `doc`. You can have fully changed the input sentence.

Of course there are other ways to change the input sentence. You can use ^input() along with ^fail(SENTENCE) to insert a replacement sentence into the input, stop processing the current one and move onto the next. That means rerunning your code all over again. Or you can use ^analyze() to revise all the NL data during your current RULE so that CS just continues the rest of your code with a new context. But they are much more expensive because they all do the complete NL process of spell checking, part-of-speech tagging, parsing, and marking. A simple ^replace is cheap.

Another use for ^replace is with pronoun resolution and entity labelling in general. If you have topics that examine all the nouns in sentences and create JSON objects describing them, then you can replace the words and any resolved pronouns, with the JSON object name. So that a sentence is transformed from:

```
I saw a brown dog. The dog ran. Then he ran faster. He saw a dog.
```

into:

```
I saw a brown jo-23. The jo-23 ran. Then jo-23 ran faster. jo-23 saw a jo-24.
```

With appropriate markings on these objects, pattern matching will do the right thing even with pronouns instead of real words, And having used ^replaceword, memorizing words will memorize their object name, making it easy for later code to interpret meaning and build a true interpretation of what happened. You can build up data in your JSON object as the story progresses. We learn that jo-23 is a dog, it's brown, it's alive, and can run. For jo-24 we know its a dog. We don't yet know its color or even if it is alive.

Yes, you could build your own structures to mirror a sentence with your data, but it will be a lot of indirection to memorize a word and then convert its position into your own structure's position to access your data.

Summary

There are lots of NL tools out there that do specific things. There are part-of-speech taggers, sentence parsers, colocation calculators, sentiment analyzers. But they provide little assistance in determining meaning.

CS is unique in allowing you to detect patterns of meaning. The engine has extensive features for supporting such detection and is designed so that you can alter most any feature to suit your needs. This flexibility means you can write bots that can detect meaning and be master of all.