ChatScript Practicum: Gleaning

‘”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.

To glean is to extract information. If there is one thing that makes a chatbot seem intelligent it is the ability to grab information from what the user says and make use of it in subsequent volleys. The very first chatbot, Eliza, fooled humans merely by grabbing critical keywords from a user’s input and spitting back precanned questions. If a user said “my wife is like my mother”, Eliza could focus on `like` and respond “In what way?” Or it could focus on the word mother and say “Tell me more about your mother”. It did not store information beyond the use for generating the reply. But our use of gleaning is to detect information and hold it around.

The process of coding gleaning takes the following steps. First, decide what you want to glean. Second, generate sample sentences containing data to be gleaned. Typically 4 or 5. Third, write script to perform the gleaning. Fourth, decide how to store the results. Fifth, find a way to make use of the gleaned data. Sixth, read logs, find new sentences you don’t handle and handle them.

What you glean is up to you. Name, hobby, illnesses, favorite fruit, whatever. Let’s consider gleaning information about a pet. You might want to glean kind of animal, name, breed, age, gender. Obviously first in line is gleaning kind of animal. To simplify things, we will start by only learning about pets of the speaker. If the user says “my friend has a dog”, we won’t care (otherwise we have to master gleaning who owns the dog other than `I`).

Gleaning animal type

We start with sample simple sentences. How might a user typically say they have a dog?

```
I own a dog.
I have a puppy.
My dog is hungry.
I own a grey cocker spaniel.
```

These would be common ways to express ownership of a dog. There are also obscure ways, and at some point we will have to throw in the towel and admit our program will not be perfect. Things like:

```
I possess a dog.
I bought a dog the other day.
I have raised this puppy from birth.
```

Taking our simple sentences, we then generate scripts to detect them. We need to decide what we are gleaning. I'm gleaning the kind of animal, in this case always a dog. `puppy` is tied to age and `spaniel` is tied to breed. The 3 key components of the pattern are detecting the animal and the evidence of ownership and evidence of 'I-ness'.

For now I will use rules that accept both statements and questions, because maybe the user will ask a question and reveal pet ownership like "What should I feed my dog?"

Sentence 1 "I own a dog."

```
u: OWN('I  own  a dog) $pet = dog
```

This covers my first sample. But it's very specific and won't handle `I own a black dog`. So I generalize it by allowing a few intervening words.

```
u: OWN('I  own *~4 dog) $pet = dog
```

I don't use the infinite wildcard `*` because I expect the word `dog` to be close. Infinite wildcards risk false detection. My pattern avoids sentences like `I own a cat because I hate dogs.` Similarly I generalize between `'I` and `own` to catch `I already own a dog`.

```
u: OWN('I  *~2 own *~4 dog) $pet = dog
```

Sentence 2 "I have a puppy"

To handle the next sample input, I need to detect `puppy` and `have`. I could write entirely new patterns, but that would become unwieldy to maintain. And someday I'll want to detect other animals and breeds. So I create a concept to hold the dogs I want to detect, and I use the preexisting ChatScript concept for ownership (or else I could define a new one). While I show the concept next to the `u:`, in reality there would be some intervening topic declaration.

```
concept: ~dogpet ( dog puppy)
u: OWN('I *~2  ~own *~4 ~dogpet) $pet = dog
```

Sentence 3 "my dog is hungry"

The third sentence requires a new pattern construction that simultaneously handles `I` and ownership.

```
concept: ~dogpet ( dog puppy)
u: OWN('I *~2  ~own *~4 ~dogpet) $pet = dog
u: MY(my *~4 ~dogpet) $pet = dog
```

Again I allow some space between `my` and `~dogpet` to support `my brown mangy dog`. From experience I know to allow for some intervening words all along. Otherwise I'd write my pattern initially without and add generalization later.

Sentence 4 "I own a grey cocker spaniel."

The fourth sentence requires detection of a dog breed. For that, we just add the corresponding CS concept to our `~dogpet`. Or invent a new one.

```
concept: ~dogpet ( dog puppy ~dog_breeds)
u: OWN('I *~2  ~own *~4 ~dogpet) $pet = dog
u: MY(my *~4 ~dogpet) $pet = dog
```

Is your glean stable?

One flaw of all of the above rules is that they can change their mind about what pet is given. Probably it's better to insure it stops after finding a pet. So I add `!$pet`.

```
concept: ~dogpet ( dog puppy ~dog_breeds)
u: OWN(!$pet 'I *~2  ~own *~4 ~dogpet) $pet = dog
u: MY(!$pet my *~4 ~dogpet) $pet = dog
```

The above cannot handle the following volley input: "I own a pet. It is a dog." To handle that would require code to manage pronoun resolution. Also doable, but not in this lesson. But this example is extremely unlikely to be seen, so too bad.

Expanding from dogs to all animals

And so we have simple detection of owning a dog. And if we want to expand this to handle other animals? We have to do two things. First, expand (and rename) our pet concept. Second, save the right kind of pet animal. This will require memorizing the animal said, and sometimes decoding it into the appropriate animal type. I use ^refine for that to keep the code clear.

```
concept: ~petlist ( ~animals puppy ~dog_breeds ~cat_breeds)
u: OWN(!$pet 'I *~2  ~own *~4 ~petlist) ^refine()
    a: ([_0==puppy _0?~dog_breeds _0==dog) $pet = dog
    a: ([_0==kitten _0?~cat_breeds _0==cat) $pet = cat
    a: () $pet = _0
u: MY(!$pet my *~4 ~dogpet) ^refine()
```

```
    a: ([_0==puppy _0?~dog_breeds _0==dog) $pet = dog
    a: ([_0==kitten _0?~cat_breeds _0==cat) $pet = cat
    a: () $pet = _0
```

Using ^reuse

Something like above would do the job. But I repeated that ^refine for the MY rule and created code that is harder to maintain. Rather than repeat the code, I would use `^reuse` like this:

```
concept: ~petlist ( ~animals puppy ~dog_breeds ~cat_breeds)
u: OWN(!$pet 'I *~2  ~own *~4 ~petlist) ^refine()
    a: ([_0==puppy _0?~dog_breeds _0==dog) $pet = dog
    a: ([_0==kitten _0?~cat_breeds _0==cat) $pet = cat
    a: () $pet = _0
u: MY(my *~4_ ~petlist) ^reuse(OWN)
```

But that won't work if I have text to also send to the user as in below, because we get the dog message as well as the cat message.

```
u: OWN(!$pet 'I *~2  ~own *~4 _~petlist) ^refine() I love dogs.
    a: ([_0==puppy _0?~dog_breeds _0==dog) $pet = dog
    a: ([_0==kitten _0?~cat_breeds _0==cat) $pet = cat
    a: () $pet = _0
u: MY(!$pet my *~4 _~petlist) ^reuse(OWN) I love cats.
```

Placeholder rules - s: LABEL(?)

To fix our problem, create a placeholder rule to ^reuse that says nothing and can't be triggered accidently.

```
concept: ~petlist ( ~animals puppy ~dog_breeds ~cat_breeds)
u: OWN(!$pet 'I *~2  ~own *~4 _~petlist) ^reuse(SAVEPET) I love dogs.
u: MY(!$pet my *~4 _~petlist) ^reuse(SAVEPET) I love cats.
s: SAVEPET(?) ^refine()
    a: ([_0==puppy _0?~dog_breeds _0==dog) $pet = dog
    a: ([_0==kitten _0?~cat_breeds _0==cat) $pet = cat
    a: () $pet = _0
```

The SAVEPET placeholder rule never triggers on its own, because it requires the input to be both a statement (s:) and a question (?). Won't happen. And the SAVEPET rule is given in _0 the matched animal from either rule calling it.

OK. We have detected one animal type. Now, do we go on to more data about it or do we work on someone who owns multiple animals? I'd move on to more data because our chatbot can appear smart merely recognizing one. And writing

patterns to detect multiple instances is more complex. And writing where to store the answer is more complex.

Gleaning animal gender

We start the same way, deciding sample sentences.

```
I have a female golden retriever.
My cow is hungry.
My cat is grey and he lives outdoors.
My cat is grey. He lives outdoors.
```

While I could stumble my way through a bunch of iterations, let me just jump to how I think the answer goes.

```
concept: ~femaleness (she her female cow)
concept: ~maleness (he him male bull)
...old code for detecting animal type ...
u: ($pet ~femaleness) $petgender = female
u: ($pet ~maleness) $petgender = male
```

The above gets us gender in most situations. We require that a pet has been detected already. While seeing a gender pronoun in a later sentence is common, one does not typically see one before the item referred to. You won't likely see "she loves milk does my cat" but you could see "my cat she does love milk"

We don't care if the gender marker has occured in the same sentence or a later one. It's a start. Then we have to think of inputs we can get fooled by. We will be fooled by inputs that involve gender but not of our pet, like `My dog was sick so my sister was sad and she said so.` For now I don't care. In advanced code we could handle this. Similarly it will be a lot of work to correctly distinguish "Harold has a cat who scratched him" and "Harold has a cat who scratched himself". So we won't bother. We will tolerate errors to get it right most of the time.

Preventing false positives

Gleaning from specific words or phrases (concept sets) is easy. The hardest thing is to avoid false positives, where the keyword doesn't mean what you think it does. There are two ways to handle this.

Using ^refine

The first is to use '^refine' and something like `@_0+` to exclude some matches.

```
u: (!$pet _~petanimals) ^refine() ^retry(RULE)
    #! My teenager is feeling puppy love -- dont react
    #! he still has puppy fat in his cheeks -- don't react
    a: (@_0+ [love fat])
    #! you'd have to be one sick puppy to want to go to the hospital -- dont react
    a: (@_0- sick)
    a: () $pet = _0
)
```

The above has the advantage of entirely local code- it's clear and you know where to modify things. It keeps scanning the sentence until $pet has be set or matches run out. It is also faster and easier to read in traces, because if the main pattern doesnt match, you never have to try all the rejoinder rules.

Using ^unmark

The alternative is to hide ~petanimals from matching. This would be good if there are several rules depending upon it. You would do that like this:

```
#! My teenager is feeling puppy love -- dont react
#! he still has puppy fat in his cheeks -- don't react
u: NOTPUPPY (_~petanimals [love fat])
    ^unmark(~petanimals _0)
    ^retry(RULE)
#! you'd have to be one sick puppy to want to go to the hospital -- dont react
u: NOTPUPPY(_~petanimals @_0- sick)
     ^unmark(~petanimals _0)
    ^retry(RULE)
```

The above hides inappropriate detections of ~petanimals, leaving only the valid ones. It is better to use ^unmark on the specific concept rather than doing `^unmark(* _0)` because making the word completely invisible can have unintended other consequences. But the above is messier, slower, uses more code, and generates more trace than this equivalent:

```
u: NOTPUPPY (_~petanimals) ^refine() ^retry(RULE)
    #! My teenager is feeling puppy love -- dont react
    #! he still has puppy fat in his cheeks -- don't react
    a: (@_0+ [love fat]) ^unmark(~petanimals _0)
    #! you'd have to be one sick puppy to want to go to the hospital -- dont react
    a: (@_0- sick)  ^unmark(~petanimals _0)
```

So mostly I think using ^refine is better than multiple top-level rules.

Gleaning names

Some things you need to glean will not show up in a concept set, or may only partially. The classic example is gleaning a name. It can be any word or sequence of words and what you usually have to detect is the context bounding the name. You have to start making assumptions about what a reasonable name will consist of. So, as always, first write sample inputs, then make patterns for them.

```
His name is Roger.
we call him rabbit.
HE IS CALLED HAROLD THE MAGNIFICENT AND IS 12.
My cat Harold likes birds.
```

So we start with the first sentence.

```
u: (name is _~noun_proper_singular)
```

The above is simple, but will fail when the system doesnt know the name. Like for "His name is pansy." So maybe we also need a context dependent rule.

```
u: NAMEIS(name _is) ^refine()
    a: CONCEPTNAME(@_0+ _~noun_proper_singular) $petname = '_0'
    a: CONTEXTNAME(@_0+  _*~3 [> , ~conjunction])  $petname = '_0
```

Note here we don't want the canonical of the name, we want the actual name, so we use quoted _0. The CONTEXTNAME pattern assumes names are usually 3 or fewer in length and are bounded by some typical delimiter like end of sentence, comma or `and`. It would fail for `Her name is Little Red Riding Hood`, unless the user is using capital letters meaningfully and proper name merging is on. We could increase the wildcard to `*~4`. But the wider we make that wildcard, the more room for mischief. "My name is something my mother liked."

And we can extend the rule to handle `called` like this:

```
u: NAMEIS(name _is) ^refine()
    #! his name is Peter
    a: CONCEPTNAME(@_0+ _~noun_proper_singular) $petname = '_0'
    #! his name is master of all.
    a: CONTEXTNAME(@_0+  _*~3 [> , ~conjunction])  $petname = '_0
#! he is called Roger
u: CALLED( is _called) ^refine(NAMEIS)
#!  we call it Roger
u: CALLED1( 'call _[him her it] ) ^reuse(NAMEIS)
```

And we can handle an appositive name like this:

```
#! my cat Harold the Great jumped from the roof
u: APPOSITIVE(~petanimals _*~3 ~verb)
```

Dealing with false positives

Gleaning is typically done from a finite set of choices. And sometimes you have to resolve issues around accepting or rejecting a match. Consider the `~country` set provided by ChatScript. It includes countries like `Turkey` and `Togo`. 'Turkey' is an issue because maybe the user is referring to the bird. So when you glean, you have to decide whether your code will try to detect the bird contexts or the country contexts. E.g.,

```
u: NOTTURKEY (_Turkey) ^refine() ^retry(RULE)
    a: (@_0- *~4 [in from live reside of])
    a: (<<Turkey [Istanbul Ankara Izmir Bursa Adana]>>)
    a: () ^unmark(~country _0)
```

Above code tries to detect Turkey used as a country, and ignore all other uses. It won't be perfect. It is probably harder to discard ones based on the bird, but here is an attempt:

```
u: NOTTURKEY (_Turkey) ^refine() ^retry(RULE)
    a: (@_0- [~determiner ~possessive]) ^unmark(~country _0)
```

```
`Togo is a different problem. A perfectly unique word, but users can easily misspell `to go`
expect the country to be referenced, so we just decided to ignore it as a country altogether
Problem is, that ships with ChatScript, so tampering with that creates confusion when you up
as a country.  Second, you could add a glean rule like this:
```

u: NOTTOGO (_Togo) ^unmark(~country _0) ^retry(RULE)

```
That kills all country uses of Togo. But it's inelegant to dedicate a rule to it.

The third trick is to define a customized concept set and use that in your code instead.
```

concept: ~ourcountries (!Togo ~country)

```
You can exclude specific items from a concept set that is inheriting items from other concep
`(![choice1 choice2) ~inheritfrom)```.

Our last example is Georgia (the state) versus South Georgia (part of the Falkland Islands)
When someone says `South Georgia` clearly it will trigger `Georgia`. Or will it?
That depends on various things. One is the state of $cs_token having proper name merging on
matched. But CS has a default ability to peek into a proper name and match the last word of
Old CS was less powerful, so it needed to peek within. Nowadays I am not fond of peeking wit
matches in phrases without having to merge the tokens. So except in rare bots, I turn off Pr
```

replace: "South Georgia" South_Georgia

```
but you also have to tell $cs_token to include #NO_WITHIN so it won't peek.

An alternative even with peek within is to let it match. Now that gets interesting. Because
it like this: `(_~country)` you may still see `Georgia` as the value (since it is a country
```

To get past this, you need to actually look at the actual value as tokenized. So you can do

u: COUNTRY (_~country) $_index = ^position(START _0) if (_0 == Georgia AND ^wordAtIndex(canonical $_index) == South_Georgia) { _0 = South_Georgia ^setposition(_0 $_index $_index) }

This code discovers that Georgia is being returned as the country in `_0`. It retrieves the It pulls that word from the sentence, and if it is actually `South Georgia`, it revises the then revises the match position of _0. It's obscure. But it fixes the problem.

So what should you do? Well, I advise against allowing peek within. Whether or not you want `replace` to make it a single token then you can easily block the Georgia part of the match

u: NOTGEORGIA (_Georgia @_0- South) ^unmark(~country _0) ^retry(RULE)

I prefer that to this rule:

u: NOTGEORGIA (South _Georgia) ^unmark(~country _0) ^retry(RULE)

because the important word to match is Georgia. If that's not there, who cares about matchin But matching the words in order makes for a clearer pattern, and CS is fast enough you may p you always match on the primary word and then refine on the secondary ones.

Extending the concept of concept

Concepts are lists of words or phrases. But sometimes you wish you could use patterns. You c

u: (my *~2 (wife husband daughter son)

The above is a classic simple sequential pattern. We can convert this into a 'concept' as fo

u: (_(my *~2 (wife husband daughter son)) ^mark(~myfamily _0)

The above rule creates a fake concept called `~myfamily` that takes up a contiguous sequence

u: (~myfamily enjoys)

The above will match "my only husband enjoys". ^mark can be applied to fake words as well, to use it to create a fake concept.

You need to remember this is a multiword pattern, in this case containing the word 'my'. So

u: (can [I ~myfamily] love)

will fail on input like this `can my daughter love pizza`. Why? Because canonical `I` will m and then `love` is not the next word (`daughter`), so matching fails. And an internal retry

Note: since these are dynamic markings, you cannot use these words inside of some other conc into a concept is the classic way to find the earliest match among all words, rather than wa

Multi-sentence Gleaning

The above stuff works fine on single sentence inputs, but how do you manage to glean multi-s
script itself. You need to leaf thru all sentences to glean, and then later decide how to re
might look like:

topic: ~main_control system repeat() ... # OOB and other stuff

save input to retrieve later

u: GLEAN() $_sentenceCount += 1 $_temp = $save Sentence(_sentenceCount) ^nofail(TOPIC ^respond(~glean_data)) if (%more) {^end(SENTENCE)}

u: OUTPUT() $_count = 0 loop(_sentenceCount) { $_count+ = 1 $restore Sentence(_count) ^respond(~realcontrol) } "' The GLEAN rule counts how many sentences there are, saves them, and gleans them until there are no more sentences. The OUTPUT rule restores each sentence and passes it off to a more normal control topic to handle the user output determination. One assumes that the GLEAN topic will have set various facts and variables to remember what it found so the ~realcontrol topic can be guided in what it does.

Understanding meaning is hard

Obviously we can devise sentences where all these things fail. Language is difficult. I mean, if I say "my feline harold who likes birds kills snakes." Is it a cat? What is the name of the cat? Is it Harold or is it some Indian name like "Harold who likes birds"? What you need to do is make assumptions and then also write code where the user can correct you when you get it wrong. And generally you are aiming to match the probable. And accept the failures of the improbable.