ChatScript Practicum: Rejoinders

 Revision 2/18/2018 cs8.1

‘”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.

Responders and rejoinders are the workhorses of an intelligent-appearing bot. They react to what the user says. Making them work requires skill in writing rule patterns and skill in designing a control script. Clever patterns are needed to detect the user’s intent. A clever control script is needed to handle cases where users input more than one sentence in a volley. But we’re not going to talk about either of those important areas in this practicum.

We’re going to talk about rejoinders and how they can be modified or enhanced with other engine functions.

Simple Rejoinders

Of responders and rejoinders, rejoinders are the best and easiest. Though they can follow any rule, generally they follow a gambit and react on the user’s immediately following input. A well-crafted gambit allows you a strong ability to predict what the user will say and therefore the rejoinder pattern can often be trivial.

Simple bots try to tightly constrain the user by asking a yes-no question, for which ChatScript has useful concept sets.

```
t:  Do you like pineapple?
    a: (~yes) Yup, it's sweet and yummy
    a: (~no) Too acidic for you, I guess.
    a: (~dunno) How can you not know?
```

Note that I indent rejoinders. This creates an easy visual structure for understanding what is happening in script.

But yes-no questions get boring and obvious. The next level of rejoinders works with a constrained set of possibilities. For example:

```
t: What natural disaster do you find most interesting?
    a: (earthquake) It is exciting to feel everything moving.
    a: (flood) Best to have a boat on standby.
    a: (fire) They get a lot of fires in California.
```

While these rejoinders would never work safely as a responder because the pattern is too prone to false matches, given the context they are very safe.

Note also that when a rejoinder matches, it completes the rule and no other rejoinders will fire, whether or not any output is generated. This may mean you want to order your rejoinders in a particular way to get the best result, just as you would with any rules in a topic.

^refine()

You can change the behavior of a rejoinder block by putting ^refine on the main rule. This changes things so that instead of waiting for the user's next input, they act immediately on the current input. There are a lot of useful characteristics from this. First, they can allow you to speed up processing and make debugging easier by reducing tracing or stepping. Consider the following sequence of gambits where the user has clicked on some Google keyword about disasters and that data has been passed into a bot:

```
t: (!$started $keyword==earthquakes) Hi. You like earthquakes? $started = 1
t: (!$started $keyword==floods) Hi. You like floods? $started = 1
t: (!$started $keyword==fires) Hi. You like fires? $started = 1
t: (!$started) Hi. What disaster are you interested in? $started = 1
```

As given above, each rule will be checked to see if it matches. When one matches that rule erases. But all the other rules will try to fire on any future input (failing because $started matches). If you've got a lot of these, the system will waste time trying to match the rule (which generally won't matter because CS is fast). But if you are tracing or stepping thru your code, you get a lot of noise. This is the perfect use of a ^refine():

```
t: ()  ^refine()
    a: ($keyword==earthquakes) Hi. What do you want to know about earthquakes?
    a: ($keyword==floods) Hi. What do you want to know about floods?
    a: ($keyword==fires) Hi. What do you want to know about fires?
    a: () Hi. What kind of disaster are you interested in?
```

This code is smaller and faster. Rejoinders are not rules that are separately erased. They automatically disappear as the rule above it erases. So when the gambit fires, it will call on some rejoinder and erase itself. We don't need $started to block the unused choices. The gambit, once erased, will not clog up tracing or stepping.

[Actually the engine never erases the code of a rule. It is merely marked as disabled for that user and its code can even be reused by other rules]

Similarly, you can do common subexpression on patterns if you want. Imagine you have a hundred responders built to answer the question 'who is xxx' where

xxx is a famous person. The efficient way to write that is:

```
?: ( < who be) ^keep() ^refine()
    a: (Mary Poppins) A babysitter
    a: (Mickey Mouse) A Disney icon
...
```

Here we want to be able to ask the same question about different persons, so we add ^keep() to keep the top rule from erasing or we add this to the topic flags so no responders in the topic erase themselves.

But the ultimate in gambits with the user will involve totally free-form questions. They are trickier to script and typically hunt for keywords and affect words.

```
t:  Why are you a vegetarian?
    a: ([ethical ethics moral principle]) That is very noble. But I like beef.
    a: ([treatment pain]) Our current animal practices are not ideal.
    a: (environment) Cows are environmentally good, when grazed properly.
    a: (~badness) I understand your complaint, I just don't agree.
    a: (~goodness) That benefit is generally overrated.
```

Nested rejoinders

One is tempted to put rejoinders on rejoinders. And if you could realistically predict the user at each step you would have a great conversation. But I've found it rarely pays to have a b: level rejoinder (unless you asked a yes-no question on the a: level.

Finding entities. . . ^retry(RULE)

When you write a bot, you probably need to both detect the intent (what the user is inquiring about) and entities, which are specific relevant information. For example for a weather bot, you need to detect that the user wants a weather report, but you also probably need to detect the location and the time involved.

Now imagine you are writing a medical bot, which needs to detect all instances of blood (an entity) in a sentence. You can ask a rule to repeat itself by using ^retry(RULE). The clever bit is that once a rule matches, it remembers where the match starts, and a retry request will execute the rule starting the pattern match after the prior match's start word.

```
u: (_blood)   ...do something with the match ... ^retry(RULE)
```

If the input is 'My blood is redder than your blood', this rule will first find blood at word 2, then restart the rule at word 3 and detect the next blood at word 7.

But some references to blood may not actually about blood itself. This use of refine combines sophisticated pattern matching with a ^refine block to only deal with the valid instances.

```
u: (_blood) ^refine() ^retry(RULE)
    a: (@_0+ pressure)
    a: (@_0- red @_0+cell)
    a: ()  ...do something with the match ...
```

The fancy @_0+ and _0- say: 'go to where the match _0 happened, and start matching either backwards (@_0-) and then forwards((@_0+). So the first a: pattern jumps to word 2 and starts matching forward at word 3, detecting 'blood pressure'. The rejoinder fires and does nothing, because we don't care about this reference. Similarly the second rejoinder hunts first immediately backward and then immediately forwards, looking for 'red blood cell' and again not being interested. The third rule accepts all other blood references and handles the data. THEN, note that when ^refine completes, control returns to the top rule and it then executes ^retry(RULE) to hunt for the next match. When, eventually, no matches are found, the top rule fails and control moves on. So here, the ^refine acts as a filter to decide which references are important and which are not.

Notice that I put the ^retry(RULE) on the top rule. If I wanted some rejoinders to retry the main rule and others not, I would put the ^retry on the rejoinder itself and say ^retry(TOPRULE). Or I can put ^retry(RULE) on a rejoinder and have it try itself multiple times locally and then to retry the main rule when it returns.

Also note that any rule whose pattern begins with a fixed location element, like < or @_0+ cannot be successfully retried because the start location cannot be moved past the match. The pattern explicitly says start here.

Shared rejoinders - ^setrejoinder()

Sometimes you want to share rejoinders across multiple rules. You can do that easily enough by placing the rejoinders under one of the rules, and having all other rules use ^setrejoinder and naming that rule.

```
u: RULE1 (...)
    a: (...)
    a: (...)
u: RULE2 (...) ^setrejoinder(OUTPUT RULE1)
```

RULE2 simply shares RULE1's rejoinders. It doesn't even matter if RULE1 has been erased already. The rejoinders are still available.

Correspondingly there is a trick to create a common dummy area to be reused...

```
u: RULE1 (...) ^setrejoinder(OUTPUT RULE3)
```

```
u: RULE2 (...) ^setrejoinder(OUTPUT RULE3)
s: RULE3(?)
    a: (...)
    a: (...)
```

Here, RULE3 holds the common rejoinders, but itself can never possibly fire because it is a contradiction. It will only be considered for firing if the input is a statement (s:) but the pattern requires it be a question.

^sequence()

Yet another function can modify the effects of a responder block. ^sequence is similar to ^refine, in that it matches on the current user input rather than the next. But ^sequence differs in that it will try every single rejoinder. Those that fire do their thing and the system will keep testing them until all are tried.

```
u: (I *~2 like _*1) ^sequence() ^retry(RULE)
    a: (_0?~fruit)  ... note fact about fruit ...
    a: (_0?~yellowitems) ... note fact about yellow items...
    a: (_0==banana)  I love bananas.
    a: (_0==grapefruit) I love grapefruit
```

In the above we are detecting the user likes something in sample sentence `I like bananas`. We want to record information and sometimes react to it. So by executing in sequence every rejoinder, we can capture that they like fruit, they like something yellow, and we can react and comment about bananas. Grapefruit is not detected. And because the user might say `I like bananas but I also like peaches`, we can capture data about all this. If we only care about one occurrence in the sentence then we just omit ^retry(RULE).

Rejoinder substitutes - ^incontext()

The thing about simple rejoinders is that you can only match one of them. But suppose you can anticipate the user asking more than one question about your gambit.

```
t: I lived in SF.
    a: (when)  2 years ago.
    a: (where) In the Presidio.
```

In the above we have two likely questions a user might ask. But we can only answer one of them. The other will fall to a responder. How do we manage this? We use ^incontext().

```
t: LIVEDSF() I lived in SF.
```

```
    a: WHENSF (when)  2 years ago.
    a: WHERESF (where) In the Presidio.
u: (when ^incontext(LIVEDSF) ^reuse(WHENSF)
u: (where ^incontext(LIVEDSF) ^reuse(WHERESF)
```

^incontext returns true when a rule by the name supplied has executed within the last 4 volleys. We could even skip the rejoinder code and put the answers on the responders like this:

```
t: LIVEDSF() I lived in SF.
u: (when ^incontext(LIVEDSF) 2 years ago.
u: (where ^incontext(LIVEDSF) ^ In the Presidio
```

Note that you can have any number of rules with the same label. This means we might even extend the life of the ^incontext by naming each the same.

```
t:  LIVEDSF() I lived in SF.
u:  LIVEDSF (when ^incontext(LIVEDSF) 2 years ago.
u:  LIVEDSF (where ^incontext(LIVEDSF) ^ In the Presidio
```

So here, if we execute the gambit, the user can ask us when and we can catch that. And maybe they spend a couple of volleys talking about the time 2 years ago. But they can still return to saying and where and we still have context to reply about the Presidio.

%more and ^next(INPUT)

Sometimes you want to try to use a rejoinder area to cover multiple sentence input. Suppose your gambit asks 'do you like fruit'. Someone might answer 'yes', someone else might answer 'I like bananas', and someone else might answer 'yes. I like bananas'. And you start to write a rejoinder to detect those cases maybe like this:

```
t: Do you like fruit?
a: (~yes !%more)  What kind of fruit?
a: (banana) I love bananas.
```

Your first rejoinder detects `yes`. And using %more can tell that there is no more input from the user in this volley, so it's safe to follow up with what kind of fruit. Your second rejoinder detects the case where they simply reply `I like bananas`. But what do you do for the third case which combines the two?

You can use ^next(input) to slide along to their second input, and then try to match that against the rejoinders. I.e.,

```
t: FRUIT() Do you like fruit?
    a: (~yes !%more)  What kind of fruit?
    a: (banana) I love bananas.
```

```
    a: (%more)  ^next(INPUT) ^refine(FRUIT)
```

The third rejoinder says since there is more input and we already have handled anything the chatbot could want about the yes input, change the system at this moment to be using the next input. So suddenly everything is set up to process rules on `I like bananas`. And then we simple tell the system to take that current input and run ^refine() on the block of rejoinders. So it can detect banana and respond. And if it can't detect anything, you just drop out of the rejoinders and continue with normal script execution.

Actually, that third rule is faulty. Can you figure out why?

If there are 3 inputs, the first of which is yes and the second of which doesnt match `bananas`, then it will execute again, lose the second input and move on to the third. That seems bad. You can easily add a flag into the pattern and the output so that it will only execute once, regardless.

Rejoinders and Control Scripts

A control script typically tests for out-of-band inputs before anything else. They are treated by CS as a separate sentence in the user's input and the control script will process it and ^end(SENTENCE). So it never gets to ^rejoinder() processing for that. But normal user sentences will typically go to rejoinders before trying to do responders or gambits anywhere. So what does it mean if your rejoinder fails to fire?

It could mean, of course that your rule pattern didn't match. Or it could mean you messed up the control script before the rejoinder call (assuming you changed the control script). One way to fail is to have a bug such that the script fails and so ends prematurely. You can protect against some of that using ^nofail(RULE). Another way to fail is to actually generate user output from earlier in your control script. After all, CS normally stops when it gets some user output.

The other thing to note is that the engine api ^rejoinder() is not something sacred. One can write script to perform custom behaviors that either use it or replace it. Remember this example?

```
t: FRUIT() Do you like fruit?
    a: (~yes !%more)  What kind of fruit?
    a: (banana) I love bananas.
    a: (%more)  ^next(INPUT) ^refine(FRUIT)
```

While it's clever, one has to write that %more rule on every rejoinder block you want it to happen. Makes for ugly code. So in my chatbot I have custom rejoinder script that does that automatically. It allows me to write simpler cleaner code like this:

```
t: FRUIT() Do you like fruit?
```

```
a: (~yes)  What kind of fruit?
a: (banana) I love bananas.
```

It get the rule tag for where the rejoinder is, it walks the rules of rejoinder block and retrieves their pattern code. If the user input is yes, it sees if the block has a ~yes handler and if it does, performs the ^next and then tests the result against all the rejoinder choices. If there's a match it performs the output of that rule. All in script, never calling the system's ^rejoinder() code. I leave it to your research to find the appropriate engine calls for this.

Summary

As a general rule, reducing rule count to improve speed is a waste of time (ChatScript is too fast). But moving rules into rejoinder areas attached to a rule pays off in debugging time (tracing and stepping) and often makes the code intent clearer.

And there are all these other interesting uses that rejoinders can be put to.