ChatScript Exotica Examples

9/17/2013 CS 3.62

^SetRejoinder

Normally the rejoinder is script immediately after the executed gambit or responder or rejoinder. But if you want to direct the system to be elsewhere, you can use SetRejoinder to do that. Below is an example wherein the normal rejoinders are set to cover a small robot's rejoinders to the dangers of getting wet.

```
t: REJOIN() If you poured coffee over me or let me fall into the bathtub what do you think w
  #! why
  a: (~why) I want you to think about and understand my vulnerabilities.

  #! get wet
  a: (wet) And then? ^setRejoinder(REJOIN)

  #! short out
  a: (short) Yes.

  #! die
  a: ([die deactivate destroy ruin]) Precisely.
```

If the human replies: *you'd die*, we go directly to *precisely*. But if the user replied *you'd get wet*, we can ask for a more thought out answer by saying *and then* and if the user then replies with *you'd short out or die* we get the app.

^SetTokenFlags

Tokenflags are set by the engine for things it detects, like question marks and exclamations. But the engine cannot detect all forms of questions and there are maybe things you want treated as questions which are actually statements.

Tell me about birds is very similar in spirit to *what do you know about birds*, for example, yet one is a statement (a command) and one is a question. If you want to extend the system marking of questions for your own purposes, you can. You would just write a topic filled with appropriate rules and call it from your control script before you do other things that might actually generate output. Below is a sample rule from such a topic:

```
#! Tell me about birds.
u: (< [~describe ~list ~explain]) SetTokenFlags(#QUESTIONMARK)
```

After this rule executes, no output is generated so it doesn't disturb other topics. But the question flag is now on, so a rule in a later topic like:

```
?: (bird) What about birds?
```

Will now react if the input is *tell me about birds*

Note that the change will not impact rule matching within the topic you have just done the change, because it has commited the set of rules it will try to match. So it only applies to later topics.

Passing in parameters

When you want to send in avatar data or other data, there is a notion of "out-of-band" communcation. The standard way (arbitrary) is to include the information at the front of the input, with `[ ]`. This can then be parsed out in various ways depending on how you want. If you are setting variables, for example, here is code for that:

```
u: (\[ _* \] _*) # grab params and main sentence - a param looks like location:100
    @0 = ^burst('_0 " ") # get all variable assigns as facts
    loop() # do a precise count
        {
        $$tmp = first(@0subject) # pick off next param
        _2 = ^burst($$tmp :) # split word and value
        _2 = ^join($ _2) # create variable
        ^_2 = _3 # do assignment onto variable
        }
    if (_1) # CS breaks off the leading [] as its own sentence. So will be null
        {
        input('_1)
        fail(sentence) # do sentence without params
        }
```

PseudoParsing:

Suppose you get input like

John is taller than Mary and Mary is taller than Sue. Who is shorter, John or Sue?

Anyone can write rules to handle that specific question. The trick is to write few rules that are general. The following esoteric use of ChatScript illustrates principles and functions for behaving a bit like a parser. The goal is to handle incoming data in one or more sentences, across one or more volleys, and locally handling pronoun references. This includes as input:

Tom is taller than Mary who is taller than Sarah. Tom is fatter than Joan but she is thinner than Harry. Tom is taller than Mary but shorter than Sarah. Tom is taller than Mary. Sarah is shorter than Tom.

The code below handles acquiring the facts (not answering the questions) but organizes the data for easy retrieval. It aims for generality. The tricks involve how ChatScript manages pattern matching. Stage one is to preprocess the input to mark in the dictionary where every word and ALL of its concept and dictionary inheritances occur in the sentence.

So if word 1 of the sentence is *tiger*, then *tiger* and *animal* and *mammal* and `~animals` and `~zoo` might all get marked as occurring at word 1. So when a pattern tries to match, it can find any of those as being at position 1. But script can also mark word positions, and it can unmark them as well. So the FACTER rule matches a series of items and after creating a fact of them, erases their mark so a rescan of the same rule can try to find a new series of items.

```
concept: ~extensions (and but although yet still)
concept: ~than (then than)

topic: ~compare_test system repeat ()

# set local pronoun
s: ( ~propername * [who she he]) refine()
  a: ( _~propername *~2 _who ) $$who = '_0
  a: ( _~femalename * _she ) $she = '_0
  a: ( _~malename * _he ) $he = '_0

# resolve pronouns
s: ( [who she he]) refine()
  a: ( _who $$who) mark(~propername _0)
  a: ( _he $he) mark(~propername _0)
  a: ( _she $she) mark(~propername _0)

s: ( _~propername * ~propername *~2 _~extensions {be} {less more} ~adjective ~than ~properna
  mark(~propername _1 ) $$and = '_0

#! Tom is more tall than Mary
#! Tom is taller than Mary and Tom is shorter than Joan.
#! Tom is less tall than Mary
#! Tom is taller than harry but shorter than Joan.
s: FACTER ( _~propername {be} {more} _{less least} _~adjective ~than _~propername )
  $$order = 1
  if (_1) { $$order = $$order * -1 } # flip order
  if ($adj)
  {
   if ($adj != _2 ) # they differ
```

```
 {
 if (query(direct_svo _2 opposite $adj ) ) # its the opposite
 {
 $$order = $$order * -1 # flip order
 }
 else {$adj = null} # accept new adjective
 }
}

# adjust pronouns
if (_0 == who) { _0 = $$who}
else if (_0 == he) { _0 = $he}
else if (_0 == she) { _0 = $she}
else if (_0 ? ~extensions) { _0 = $$and}

if (_2 == who) { _2 = $$who}
else if (_2 == he) { _2 = $he}
else if (_2 == she) { _2 = $she}

if (!$adj)
{
 $adj = _2
 if ($$order == 1) { ^createfact(_0 $adj _3)}
 else {^createfact(_3 $adj _0)}
}
else # already have an adjective to run with
{
 if ($$order == 1)
 {
 ^createfact(_0 _2 _3)
 }
 else
 {
 ^createfact(_3 $adj _0)
 }
}

unmark( ~propername _0)
unmark(~adjective _2 )
unmark(~propername _3 )
$$who = _3
retry()
```

A responder for handling questions given 2 people is:

```
#! who is taller, Tom or Harry?
#! Of Tom and harry, who is taller?
```

```
#! who is less tall, Tom or Harry?
#! who is the taller of Mary and Harry
#! Of Tom and harry, who is least tall?
?: COMPARE ([which who what] be {the} {more most} _{less least} _~adjective < * _~propername
  # 0=less 1=adj 2 = person1 3 = person2
  $$order = 1
  if (_0) { $$order = $$order * -1 } # flip order
  if ($adj)
  {
   if ($adj != _1 and query(direct_svo _1 opposite $adj ) ) # they differ
   {
   $$order = $$order * -1
   }
  }

  # find if _3 is more than _2
  nofail(RULE eval(query(direct_vo ? $adj _2))) # who is taller than _2
  _7 = @0subject
  loop()
  {
   if (_7 == _3) {FAIL(rule)} # now matched
   query(direct_vo ? $adj _7) # who is taller than this
   _7 = @0subject
  }

  if ( $$order == 1) # normal order
  {
   if (@0subject) {_3 is.}
   else {_2 is.}
  }
  else # inverse order
  {
   if (@0subject) {_2 is.}
   else {_3 is.}
  }
```

The system builds facts in a specific order, like (X taller Y) and if a shorter comes first it flips the order and rewrites using common adjective notation.