ChatScript Control Scripts

Revision 9/24/2017 cs7.55

Control scripts are an advanced topic you should read about ONLY after you have read the Advanced User Manual. In fact, may you never have any need to read this manual at all. What comes shipped with by default with ChatScript may well be all you ever need.

But actually, a control script is nothing more than a topic written in ChatScript, much like any other topic. It is the topic that controls how the ChatScript engine works, which means you can write bots to operate any way you choose. Want to have your bot simulate ingesting alcohol, getting drunk, and recovering over time? You can write control scripts to do that. My Suzette bot had a control script that would analyze your interactions with her. If you were friendly, agreeable, interested in the conversation, she got friendlier.

Until she reached a point where she went neurotic and felt she wasn't deserving of your attention. This became clear in her conversations. Likewise if you were hostile, disagreeable, and bored with her, she drifted into paranoid. She wondered who might be listening in on the conversation, why you asked the questions you did. She even began to speculate how she might kill you. It's all in the control script she had (and topics she had that got invoked to simulate paranoia or neurotic affection).

The default control script shipped with ChatScript reacts to only the first sentence of your input. One can instead write a control script to react to each input sentence. It's a choice.

The engine dictates that when a rule executes and generates user output AND doesn't fail after that, then rule processing in a topic ends and so on up the call chain of topics.

The engine also has built in is that it will try to execute a control script at the start of a volley (preprocessing), a control script per sentence in the current volley, and a control script at the end of a volley (postprocessing).

Bot Macro

Actually, the first part of controlling your bot is the outputmacro that defines what happens when your bot gets first created for a user. This is where you tell the system what topics to call for the 3 control phases and what variables you want to initialize or facts you want to create. Also what topic to start out in, if any. And any controls on how input is processed and how output is processed. E.g.,

```
outputmacro: Thomas()
  ^addtopic(~introductions)
  $cs_control_pre = ~xpre_control
  $cs_control_main = ~xmain_control
  $cs_control_post = ~xpost_control
  $girlfriend = Mary
  $cs_token = #DO_INTERJECTION_SPLITTING | #DO_SUBSTITUTE_SYSTEM |
              #DO_NUMBER_MERGE | #DO_PROPERNAME_MERGE |
              #DO_CONDITIONAL_POSTAG | #DO_PARSE | #DO_SPELLCHECK
```

If you don't define `$cs_control_pre` or `$cs_control_post`, they just don't invoke anything. You need `$cs_control_main` to respond to an input.

I use `$cs_control_pre` to initialize some variables before handling each sentence. I use `$cs_control_post` to have the system analyze the chatbots own output for things like pronoun resolution and current mood, and to generate special out-of-band messages to control an avatar.

These topics should be flagged SYSTEM so they do not erase itself as they gets used.

The `^addtopic(~introductions)` is needed if you expect the bot is going to have the first word. Typically the user "logs in" and the bot responds with a greeting message first, be it *Welcome to ChatScript* or *Welcome back* if this is not your first conversation. Since there is no user input on the login message, there are no keywords to initiate a topic. One could put the starting topic as part of the control script, but it's easy enough just to say "start here" using addtopic.

The $cs_token assignment tells the system how to preprocess input (parsing, pos-tagging, spelling correction, etc).

Basic Control Scripts

The interesting thing about the pre and post control scripts, is that they are always invoked in gambit mode (hence you use t: rules). They don't have any user input to respond to, so they can't be invoked in responder mode. The main program, as it is responding to user sentences, is always invoked in responder mode and will generally consist of `u:` or `s:` or `?:` rules.

What you do in the control script is up to you. For Harry, for example, one of the things it does is call engine code for a random topic to gambit from when it has nothing better to do. But this means all topics are available to it equally. In a commercial product, the priority of topics was specified in a list. Movies, music, food are common high-priority topics. Funeral customs was a low priority topic. So we didn't want it randomly coming up early and the control script

walked the list of topics in priority order trying to find a gambit from one of them.

Single output regardless of input sentence count

The Harry bot main control script has a lot of `^if` statements of the form:

```
topic: ~maincontrol system repeat ()
    u: () if ( %response == 0 ) {nofail(TOPIC ^rejoinder())}
          if ( %response == 0 ) {nofail(TOPIC ^respond($$currenttopic))}
          ...
```

If you provide the input *my life is fun. What is the color of the sun?* the Harry bot would only respond to my life is fun and ignore the 2nd sentence. This is because the if test shown passes only if no output has been generated yet. Once output has been generated for the first sentence, the system will not try to generate output for the second sentence.

Multiple outputs, one per input sentence

A different style of control script is used by Rose, to allow multiple responses.

```
topic: ~maincontrol system repeat ()
    u: () $_response = %response
          if ( %response == $_response  ) {nofail(TOPIC ^rejoinder())}
          if ( %response == $_response ) {nofail(TOPIC ^respond($$currenttopic))}
          ...
```

This allows the system to generate one response per sentence of the input.

Meanwhile the nofail clause is because it is conceivable you write a topic that actually fails as a topic somehow. Not normal, but possible. If it did fail, it will kill the entire rest of the control topic by failing and you'd get no output. So, as a precaution against a topic written incorrectly, the safe thing to do is wrap the call to a topic or rejoinder or whatever in a `^nofail(TOPIC ...)` to protect against that. It's not required, and usually not necessary. But it's a real pain when things go sour and a topic actually fails. Merely failing to match any rules is not considered a topic failure by ChatScript. You generally have to explictly fail the topic yourself in script.

The Harry control topic has a common flow of control and you should read through that. Because control scripts are just topics and have all the richness of ChatScript available to them, you can do really weird things if you want to.

Alternative style

An alternative to the series of `if` statements in a single rule of the control script is to use a series of responders.

```
topic: ~maincontrol system repeat ()
    u: () ^nofail(TOPIC ^rejoinder())
    u: () ^nofail(TOPIC ^respond($$currenttopic))
          ...
```

This takes advantage of ChatScript's default behavior of not executing rules later in a topic if a rule generates output earlier. And since that behavior only applies while in a topic, it automatically allows an output for every input sentence.

Fancy control script

The above styles always react to the sentence immediately at hand. But suppose you wanted to read through every sentence, gather data, and then figure out how to respond.

Technically, there are two ways to accomplish that. You could set a flag indicating you are doing a first pass, process the sentence through main control (avoiding generating output). When there are no more sentences (!%more $$pass1), then turn off the first pass flag and do ^retry(INPUT), to cause the entire input to be submitted again and on this pass you react. But that's clumsy and inefficient since it forces every sentence to be analyzed twice. The efficient way is this:

```
topic: ~maincontrol system repeat()
u: (_* )
    $$sentenceCounter += 1
    $_tmp = ^saveSentence($$sentenceCounter)
    ^respond(~goanalyzethesentence)
    if (%more) ^end(SENTENCE) # go get next sentence, returned to maincontrol again

# review his inputs now to generate output...
u: ()
    $_count = 0
    loop($$sentenceCounter)
    {
        $_count += 1
        ^restoreSentence($_count)
        ^respond(~actuallygoforoutput)
    }
```

Conversation start

ChatScript can be aware of when a user first converses with the bot because there is no pre-existing topic file in the USERS folder. It initializes the input volley count to 0 (%input). Thereafter, this count automatically increases for every volley. But ChatScript does not monitor time. A user may start a conversation, walk away, come back days later, and it can still be the same conversation (though your script could determine how long it's been). But when you close down a browser and bring it back up again, in effect you are saying this is a new conversation (though an old relationship). The start of a new conversation is signalled by a null message from the browser. And in your control script, if you want to react to the start of a new conversation, you would put code like this:

```
u: ( %input<%userfirstline) ^gambit(~introductions)
```

The system knows when the new conversation started, and the input volley count will not have risen yet because the user has provided no input. That's the definition of a start message.

Input OOB

Serious applications will use OOB to pass application-specific data into and out of ChatScript. OOB data is always the first sentence of any input and is contained with [ ]. If you know your app is always providing OOB input, then there is no possibility the user can spoof the system by making their input look like OOB. You want to process OOB differently from user input, so your control script should look like this:

```
topic: ~maincontrol system repeat()
u: (< \[ )
    ^respond(~oobhandler)
    ^end(SENTENCE)

u: () # whatever normal processing you do for user sentences
```

Output OOB

The primary rule for output OOB is that it is within [ ] and is the first thing in the output. You can write rules that directly embed OOB with the user message like this:

```
u: () \[ action=wave \] How are you?
```

But that gets messy and if you output multiple messages, you cannot easily combine distinct OOB messages into a single composite OOB. Therefore the usual

technique is to define a macro that holds OOB data and use a postprocessing topic to output it at the end.

```
outputmacro: ^OOB(^value) $oob = ^join($oob " " ^value)

topic: ~introductions keep repeat ()
u: () ^OOB(action=wave ) How are you?

topic: ~postprocess system repeat()
t: ( $oob ) ^postprintbefore( \[ $oob \] )
```

In the above code use $$oob instead of $oob (pdf converter didnt like it)

Changing input token processing

There are times you may want to alter input processing by changing $cs_token. For example if you ask the user their name, on the next user input you probably don't want spell checking happening which could make a mess of foreign names (or even normal English ones CS is not aware of). So you want a convenient way to temporarily change $cs_token. Here is how:

```
# in your bot definition make a copy of your normal $cs_token
outputmacro: yourbot()
    $cs_token =  hashDO_INTERJECTION_SPLITTING
    $cs_token |=   hashDO_SUBSTITUTE_SYSTEM
    $cs_token |=   hashDO_SPELLCHECK
    $cs_token |=  hashDO_PARSE
    $std_cstoken = $cs_token

# in your script set a variable for what you want to happen next input
t: What is your name?  $$newtoken = $cs_token - hashDO_SPELLCHECK

# in postprocessing make the change over
topic: ~postprocess system repeat()

t: ($cs_token!=$std_token) $cs_token = $std_token
t: ($$newtoken) $cs_token = $newtoken
```