ChatScript Advanced User's Manual

Revision 8/31/2019 cs9.7

ADVANCED OUTPUT

Committed Output

Simple output puts words into the output stream, a magical place that queues up each word you write in a rule output. What I didn't tell you before was that if the rule fails along the way, an incomplete stream is canceled and says nothing to the user. For example,

```
t: I love this rule. ^fail(RULE)
```

Processing the above gambits successively puts the words *I*, *love*, *this*, *rule*, . into the output stream of that rule.

If somewhere along the way that rule fails (in this case by the call at the end), the stream is discarded. If the rule completes and this is a top level rule, the stream is converted into a line of output and stored in the responses list.

When the system is finished processing all rules, it will display the responses list to the user, in the order they were generated (unless you used `^preprint` or `^insertprint` to generate responses in a different order). If the output was destined for storing on a variable or becoming the argument to a function or macro, then the output stream is stored in the appropriate place instead.

I also didn't tell you that the system monitors what it says, and won't repeat itself (even if from a different rule) within the last 20 outputs. So if, when converting the output stream into a response to go in the responses list, the system finds it already had such a response sent to the user in some recently earlier volley, the output is also discarded and the rule "fails".

Actually, it's a bit more complicated than that. Let's imagine a stream is being built up. And then suddenly the rule calls another rule (`^reuse`, `^gambit`, `^respond`). What happens? E.g.

```
u: ( some test ) I like fruit and vegetables. ^reuse(COMMON) And so do you.
```

What happens is this- when the system detects the transfer of control (the `^reuse` call), if there is output pending it is finished off (committed) and packaged for the user. The current stream is cleared, and the rule is erased (if allowed to be). Then the `^reuse()` happens. Even if it fails, this rule has produced output and been erased.

Assuming the reuse doesn't fail, it will have sent whatever it wants into the stream and been packaged up for the user. The rest of the message for this rule now goes into the output stream *and so do you* and then that too is finished off and packaged for the user. The rule is erased because it has output in the stream when it ends (but it was already erased so it doesn't matter).

So, output does two things. It queues up tokens to send to the user, which may be discarded if the rule ultimately fails. And it can call out to various functions. Things those functions may do are permanent, not undone if the rule later fails.

There is a system variable `%response` which will tell you the number of committed responses. Some code (like Harry's control script) do something like this:

```
$_response = %response
...
if ($_response == %response) {...}
```

which is intented to mean if no response has been generated so far, try the code in the ^if. But you have to be wary of the pending buffer. Calling some code, even if it fails, may commit the pending buffer. If there is a chance you will have pending output, the correct and safe way to code this is:

```
^flushoutput()
$_response = %response
...
if ($_response == %response) {...}
```

`^flushoutput` will commit pending output.

Output cannot have rules in it

An output script cannot embed another rule inside it. Output is executed during the current volley whereas rules (like rejoinder rules) may be executed in a different volley. Therefore this is illegal:

```
u: GREETING ( ~emohello )
   if ($username)
   {
     Hi  $username!
   }
   else
   {
     I don't believe we've met, what's your name?
     a: (_*) So your name is '_0?
   }
```

and needs to be written like this:

```
u: GREETING ( ~emohello )
```

```
if ($username)
{
  Hi  $username!
}
else
{
  I don't believe we've met, what's your name?
}
a: (_*) So your name is '_0?
```

Of course you don't want the rejoinder triggered if you can from the `if` side, so you'd also need to add a call to `^setnorejoinder` from inside it.

Formatted double quotes (Active/Format String)

Because you can intermix program script with ordinary output words, ChatScript normally autoformats output code. But programming languages allow you to control your output with format strings and ChatScript is no exception.

In the case of ChatScript, the active string `^"xxx"` string is a format string. The system will remove the `^` and the quotes and put it out exactly as you have it, except, it will substitute variables (which you learn about shortly) with their values and it will accept `[ ] [ ]` choice blocks. And will allow function calls. You can't do assignment statements or loops or if statements.

```
t: ^"I like you."
```

puts out *I like you.*

When you want special characters in the format string like `"`, you need to backslash them, which the format string removes when it executes. For example:

```
u: () $tmp = [ hi ^"there \"john\"" ]
```

the quote inside the format string need protecting using `\"`.

You can write `\n`, `\r`, `\t` and those will be translated into newline, carriage return, and tab. However, you should avoid `\r` because on LINUX it is not needed and in Windows the system will change `\n` to carriage-return and newline.

Format strings evaluate themselves as soon as they can. If you write:

```
u: () $tmp = ^" This is $var output"
```

then `$tmp` will be set to the result of evaluating the format string. Similarly, if you write:

```
u: () $tmp = ^myfunc(^" This is $var output")
```

then the format string is evaluated before being sent to `^myfunc`.

You can continue a format string across multiple source lines. It will always have a single space representing the line change, regardless of how many spaces were before or after the line break. E.g

```
^"this is my life" # -> ^"this is my life"
```

regardless of whether there were no spaces after is or 100 spaces after is. You may not have comments at the ends of such lines (they would be absorbed into the string).

While an active string tries to detect and substitute variables, it can't succeed if you have letters immediately after the variable name. E.g.

```
^"T$$time_computed_hrs:$$time_computed_mins:$$time_computed_secsZ"
```

The Z at the end of the `$$time_computed_secs` will make that variable hard to detect, since it will look like the variable is `$$time_computed_secsZ`.

You can fix that by escaping the next character not part of the name. `$$time_computed_secs\Z`.

While you can have function calls inside an active string, the system will strip a single space in front of the call. E.g., if ^mycall returns `test`

```
$_var = ^"this is ^mycall(x)"
```

then the above sets $_var to `this istest` and you need to add an extra space in front of the call if you want a space. This is because you cannot do without the space in front of the function call merely to recognize the call (lest it join as part of the prior token). And you may want to merge the result together. Hence extra space if you want one.

Json Active Strings

`^' xxxxxx '` is another kind of active string. It is intended for writing easy JSON, because you don't have to escape doublequotes unless json would need to.

It will converting `\n`,`\t`, `\r` into their control characters, convert `\\` into just `\`, and leave all other characters alone. Eg

```
^'{ "test" : "my \"value\" \x" }
```

will become

```
^'{ "test" : "my \"value\" \x" }
```

Functional Strings

Whenever format strings are placed in tables, they become a slightly different flavor, called functional strings. They are like regular output- they are literally

output script.

Formatting is automatic and you get to make them do any kind of executable thing, as though you were staring at actual output script. So you lose the ability to control spacing, but gain full other output execution abilities.

They have to be different because there is no user context when compiling a table. As a consequence, if you have table code that looks like this:

```
^createfact( x y ^" This is $var output")
```

the functional string does NOT evaluate itself before going to createfact. It gets stored as its original self.

We will now learn functions that can be called that might fail or have interesting other effects. And some control constructs.

Loop Construct - loop or ^loop and jsonloop

Loop allows you to repeat script. It takes an optional argument within parens, which is how many times to loop. It executes the code within `{ }` until the loop count expires or until a `FAIL` or `END` code of some kind is issued.

- `End(loop)` signals merely the end of the loop, not really the rule, and will not cancel any pending output in the output stream.
- `Fail(LOOP)` will terminate both loop and rule enclosing. All other return codes have their usual effect.

Deprecated is `fail(rule)` and `end(rule)` which merely terminated the loop.

```
t: Hello. ^loop (5) { me }

t: ^loop () { This is forever, not. end(LOOP)}
```

The first gambit prints *Hello. me me me me me.* The second loop would print forever, but actually prints out This is forever, not. because after starting output, the loop is terminated.

Loop also has a built in limit of 1000 so it will never run forever. You can override this if you define `$cs_looplimit` to have some value you prefer.

^loop(n)

Loop can be given a count. This can be either a number, function call that results in a number, or you can use a factset id to loop through each item of the factset via

^loop (@0)

^jsonloop($jsonvar $_val1 $_val2)

See ^jsonloop in Json manual for looping through a JSON object or array.

If Construct - if or ^if

The if allows you to conditionally execute blocks of script. The full syntax is:

```
if ( test1 )
    {
      script1
    }
else if ( test2 )
    {
      script2
    }
# ...
else
    {
      script3
    }
```

You can omit the else if section, having just if and else, and you can omit the else section, having just if or if and else if. You may have any number of else if sections. The test condition can be:

- A variable - if it is defined, the test passes
- `!` variable - if it is not defined, the test passes (same as relation variable `==` null)
- A function call - if it doesn't fail and doesn't return the values 0 or false, it passes
- A relation - one of `== != < <= > >= ? !?`

For the purposes of numeric comparison (`< <= > >=`) a null value compared against a number will be considered as 0.

You may have a series of test conditions separated by `AND` and `OR`. The failure of the test condition can be any end or fail code. It does not affect outside the condition; it merely controls which branch of the if gets taken.

```
if ($var) { } # if $var has a value
```

```
if ($var == 5 and foo(3)) {} # if $var is 5 and foo(3) doesn't fail or return 0 or false
```

Comparison tests between two text strings is case insensitive.

A word of warning on the `?` (in set) relation test. It only works for actual concepts that have enumerated values. A number of sets marked by the engine for patterns do not consist of enumerated members.

All of the pos-tagging and parse-related concepts are like this, so you cannot use `~number`, `~noun`, `~verb`, etc here.

It will work if you compare a match variable derived from input, because that has access to knowing all the marked concepts of that particular word.

Pattern If

An alternative If test condition is the pattern If. You write the test using the word pattern at the start, and then you write exactly what you can write when you write a rule pattern. Eg.

```
if (pattern bingo _*1 ~helo) { ... }
```

This gives you the full power of the pattern matcher, including the ability to match and memorize from the current input.

Quoting

Normally output evaluates things it sees. This includes `$user` variables, which print out their value. But if you put quote in front of it, it prints its own name. `'$name` will print *$name*.

The exception to this rule is that internal functions that process their own arguments uniquely can do what they want, and the query function defines `'$name` to mean use the contents of `$name`, just don't expand it if it is a concept or topic name as value.

Similarly, a function variable like `^name` will pretend it was its content originally. This means if the value was `$var` then, had $var been there in the output originally, it would have printed out its content. So normally `^name` will print out the contents of its content. Again, you can suppress with using `'^name` to force it to only print its content directly.

Outputting underscores

Normal English sentences do not contain underscores. Wordnet uses underscores in composite words involving spaces.

ChatScript, therefore has a special use for underscores internally and if you put underscores in your output text, when they are shipped to the user they are converted to spaces.

This doesn't apply to internal uses like storing on variables. So normally you cannot output an underscore to a user. But a web address might legitimately contain underscores. So, if you put two underscores in a row, ChatScript will output a single underscore to the user.

Response Controlinput

Having said that CS automatically changes underscores to spaces, you can alter this and other default response output processing. The variable `$cs_response` can be set to some combination of values to alter behavior. The default value is

```
$cs_response = #RESPONSE_UPPERSTART +
               #RESPONSE_REMOVESPACEBEFORECOMMA +
               #RESPONSE_ALTERUNDERSCORES +
               #RESPONSE_REMOVETILDE +
               #RESPONSE_NOCONVERTSPECIAL
```

which controls automatically up-casing the first letter of output, removing spaces before commas, and converting underscores to spaces, removing ~ from concept names, and not converting carriage return, newline, and tabs from CS backslashed notation to actual ASCII character.

Equivalently

```
$cs_response = #ALL_RESPONSES
```

which if you want all is what you should use in case new ones are added in the system later.

Output Macros

Just as you can write your own common routines (functions) for handling pattern code with patternmacro:, you can do the same for output code.

```
Outputmacro: name (^arg1 ^arg2 ...)
```

and then your code. Only now you use output script instead of pattern stuff. Again, when calling the macro, arguments are separated with spaces and not commas.

Whereas most programming language separate their arguments with commas because they are reserved tokens in their language, in ChatScript a comma is a normal word. So you separate arguments to functions just with spaces.

```
?: ( hi) ^FiveArgFunction( 1 3 my , word)
```

Outputmacros can return a value, just like a normal function. You just dump the text as you would a message to the user.

```
outputmacro: ^mymac()
tested here

TOPIC: ~patterns keep repeat []
#! what time is it?
u: ( << what time >> ) $test = ^mymac() join(ok $test)
```

will print *oktested here*.

However, it is clearer and cleaner if you are returning data to be stored somewhere else (not to be merely immediately sent to the user), to use `^return("tested here")`. This both creates the result, and ends the function immediately even if other code follows.

Note - calls to macros use "pass by reference", so the actual value of the `^variable` is the name of what was passed in, and it is generally (but not always) evaluated on use.

You may make references to outputmacros before they are defined, EXCEPT when the Function is directly or indirectly referenced from a table. Tables immediately execute as they are compiled, and you will get an error if a function it tries to use is not defined.

Indirect function calls

You can store an outputmacro name on a variable and then call that indirectly.

```
^$_xx(value1)  if $_xx holds a function name
```

Sharing function definitions

ChatScript requires that a function be defined before use. When you use that function from multiple files, you may have trouble ordering the files for compilation if you merely name the folder in `filesxxx.txt` since you cannot guarantee compilation order unless you explicitly name the files. But you can also just put your functions in a top level file and then have your other files in folders, and name it and then them in your `filesxxx.txt` file.

Save-Restore locals

`$$xxx` and `$xxx` variables are global, merely transient and permanent.

Function variables like `^myval` are restricted in use to the function declaring them, so they are sort of local variables, but they are stand-ins for the arguments passed, which means if you write indirectly the function variable you are changing something above you as well.

Fortunately there are local variables, `$_xxx`.

Without local variables, it is easy to accidentally reuse the same name of a transient variable that you used above you in the call chain. Imagine this:

```
outputmacro: ^mycall()
    $$counter = 0
```

```
    loop()
        {
        # ...
        $$counter += 1
        }
```

and this:

```
$$counter = 0
loop()
    {
    $$tmp = ^mycall()
    # ...
    $$counter += 1
    }
```

You have two areas using the same counter variable and the inner one destroys the outer one. Here is where save-restore variables come in.

You can either use local variables, when you don't need to pass information between places except via function args. Or you can use save-restore variables.

You can declare a list of variables whose contents will be memorized on entry to a function or a topic, and restored to their former values on exit.
You can safely write all over them within the function or topic, without harming a caller. And they are still global, in that they are visible to anyone your function calls.

Of course if you intend to pass back data in a global variable, don't put it in your save-restore list.

```
Outputmacro: ^myfunc(^arg1)($$tmp  $global  $$tmp2) # $$bestscore exported
# code

Topic: ~mytopic(keyword1 keyword2) ($$tmp $$global $$tmp2)
# rules
```

You can protect both transient and permanent variables, but usually you would just protect all of the transient variables you assign values to inside your function or topic. The comment is what I would say if I intended a variable be returned outside in addition to a primary return value. That way anyone reading the code would know $$bestscore was not accidentally left off the save-restore list.

And whenever you can, prefer local variables because then you don't have to remember to add them to the protected save-restore list. ChatScript does that automatically.

```
Outputmacro: ^myfunc(^arg1)($$tmp $global $$tmp2) # $$bestscore exported ... code
```

You can protect both transient and permanent variables, but usually you would just protect all of the transient variables you assign values to inside your function.

The comment is what I would say if I intended a local variable be returned outside in addition to a primary return value. That way anyone reading the code would know `$$bestscore` was not accidentally left off the save-restore list.

`dualmacro: name(...)`

is exactly like `outputmacro:`, but the function can be called on then side or on the output side. Typically this makes most sense for a function that performs a fixed `^query` which you can see if it fails in pattern side or as a test on the output side or inside an if condition.

Output macros can be passed system function names and output macro names, allowing you to indirectly call things. E.g.

```
outputmacro: ^indirect(^fn ^value)
    $$tmp = ^fn(^value)
```

The above will evaluate `^fn`, and if it finds that it is a function name, will use that in a call. The only tricky part is creating the name of the function to call in the first place.

If you just write a function name with ^ in output, the compiler will complain about your call. So you have to synthesize the name somehow. Here are two ways:

```
outputmacro: ^mycall()
    $$tmpfn = ^join( ^"\^" func)
    ^indirect($$tmpfn 34)
    ^indirect( ^"\^func" 34)
```

You can also store function names on user and match variables and then call them. E.g.

```
$$tmp = ^"\^func"
$$tmp(34)
```

You can declare an outputmacro to accept a variable number of arguments. You define the macro with the maximum and then put "variable" before the argument list. All missing arguments will be set to null on the call.

```
outputmacro: ^myfn variable (^arg1 ^arg2 ^arg3 ^arg4)
```

Output Macros vs `^reuse()`

An outputmacro is a block of code, treated as a kind of function. But another way to make a block of code is create a rule and `^reuse` it. E.g.

```
s: MYCODE ( ? ) here is a block of code that goes on and on
```

Code you write in an output macro could be code you write on the output side of a `^reused` rule. Notice that this rule can never trigger on its own (an input sentence cannot be a statement and a question simultaneously).

So what are the distinctions between the two? The distinction is not in tracing. You can trace a single rule or an outputmacro equally. And both return to their caller when normally complete.

An advantage for outputmacros is that you can pass them arguments, making them more easily tailorable. To do the same with a rule, you have to store values on globals, which is more inconvenient, harder to understand, and subject to the risk that you accidentally reuse the same variable in something the rule calls. Similarly outputmacros can return a value, while you have to use globals to return a value from a rule.

An advantage for rules is that they can have rejoinders. So if the rule generates output, it may also have rejoinders to react to it. Another advantage for rules is that you can terminate their execution early `^end(RULE)` without impacting the calling rule. There is no such ability in an outputmacro, so you'd have to organize if statements to manage early termination effects.

Randomized Output Revisited `[ ]`

Remember this construct:

```
?: ( hi ) [How are you feeling?][Why are you here?]
```

These choices are picked equally. But maybe you don't want some choices. You can put an existence test of a variable at the start of a choice to restrict it.

```
?: ( hi ) [$ready How are you feeling?][Why are you here?]
```

In the above, the first choice is controlled by `$ready`. If it is undefined, the choice cannot be used. You can also use negative tests.

```
?: ( hi ) [!$ready this is a][This is b]
```

In the above only if `$ready` is undefined can you say *this is a*

If you want the variable to be a lead item in the actual output of a choice, you can do this:

```
?: (hi) [^eval($ready) is part of the output]
```

or the more clunky:

```
?: ( hi ) _0 = $ready [ _0 is part of the output]
```

Choices lead to issues when you want rejoinders. You can label rejoinder branches of choices. Those without labels default to `a:`

```
?: ( what meat ) [c: rabbit ] [e: steak] [ h: lamb] [pork]
    a: this rejoinders pork
    c: this rejoinders rabbit
    e: this rejoinders steak
    f: this rejoinders on e:
    h: this rejoinders lamb
```

In the above, pork rejoinders at `a:`, while the other choices name their rejoinder value. Each new starting label needs to be at least one higher than the rejoinder before it. That allows the system to detect rejoinders on rejoinders from choice branches.

If you do both variable control and rejoinder label, the control comes first and label after you have successful control.

```
?: ( what meat ) [$ready c: rabbit ] [e: steak] [ g: lamb] [pork]
```