ChatScript Practicum: Control Flow

‘”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.

The backbone of any program is its control flow. The basic control flow of most computer languages is sequential flow, with options for conditional flow and loops. ChatScript defines a topic as a sequential flow of rules (either gambit or responder). And it defines outputmacros in the typical way as a sequential flow, with options for conditional flow (IF) and loops (LOOP).

IF/THEN/ELSE

Technically ^if is a predefined special syntax. The compiler will accept `if` without the ^ as long as you provide parenthesized arguments (the if conditions) afterwards. The basic syntax allows for (but does not require) `elseif` and `else` clauses.

```
if (...) {...}
else if (...) {...} # optional
else {} # optional
```

```
## IF condition syntax

There are two possible syntaxes for the conditions of an if. The original syntax allowed ope
tests separated by  `AND` or `OR` like this:

if ( $foo < 5 AND $_x ? ~myset)

Wheneever there is a relational operator, you need to use spaces around it.  This contrasts
are embedded without spaces in a composite token. In patterns, the AND condition is represer
whereas the OR condition is represented using `[]` construction.
The same conditions of above, in a pattern. look like this:

u: ($foo<5 $_x?~myset)

More recently ^if statements are allowed to use pattern notation, merely by saying `PATTERN`

if (PATTERN $foo<5 $_x?~myset)


So why use one notation over the other?  In fact, the most versatile notation is the `PATTER
to create hierarchies of precedence, you can do pattern matches in existing user input and y
```

perhaps you should always use PATTERN notation. The other notation is merely historical.

IF condition and function failures

The other interesting thing about the IF condition is that it automatically traps any rule

```
if (^substitute(character $_value x y FAIL)) {}
```

So if in the above there is no x in $_value, then the fail request argument causes substitut

Loop

Simple loops execute code over and over again like this:

```
loop()
{
    ....
}
```

But ChatScript doesn't want to risk an infinite loop, so in the absence of any explicit loop control, it defaults to a liit of 1000. You can change This default by setting a value on $cs_looplimit if you want to extend it.

```
$cs_looplimit = 10000
loop()
{
    ....
}
$cs_looplimit = null        # back to default of 1000
```

You can more precisely control the loop by providing a value as argument to the LOOP

```
loop($_mycount)
{
    ....
}
```

or

```
@0 = ^query(...)
loop(^length(@0))
{
    ....
}
```

However, any such value will be forced to be no more than the max loop limit.

Other ways to end a loop involve the loop detecting a failure.

```
@0 = ^query(...)
```

```
loop()
{
    $_value = ^first(@0subject)
    ....
}
```

The above loop will execute until the ^first function fails, which will end the loop (but not the rule or the topic).

The same would be true of ^last(@0subject) or ^next(FACT @0subject)

```
@0 = ^query(...)
loop()
{
    $_value = ^next(FACT @0subject)
    ....
}
```

QUERY idiosyncracies

You have to be really careful retrieving values of a query in a loop, if you care about ordering of the values. You need to understand how facts are found by a query. When facts are created, they are added to lists associated with the field values.

```
$_tmp = ^createfact( x y 1)
$_tmp = ^createfact( x y 2)
```

The first fact is created and cross referenced on a subject list of x, a verb list of y, and an object list of 1. The next fact acts similarly, but adding to lists is the simple add to head of list. That means the subject list for x has the second fact first and the first fact second. ^query walks one of these lists (depending on the query) to get facts to consider. When you loop thru them, if you want them in order of creation, you next to use ^last to get the oldest fact. ^first and ^next will always get the most recent facts first.

Expecting a failure to terminate a loop DOES NOT WORK with JSON arrays, because JSON data accesses don't fail, they just return null when they run out. So below will execute the full loop limit as long as $_array is a JSON array, whether or not it has any values and whether or not those values are JSON objects with a name field.

```
$_count = 0
loop()
{
    $_value = $_array[$_count].name
    $_count += 1
    ....
```

```
    }
```

You can handle this using ^length as follows:

```
    loop(^length($_array))
    {
        $_value = $_array[$_count].name
        $_count += 1
        ....
    }
```

JSON idiosyncracies

One unusual side effect of how JSON data is represented is that if you know a unique name of a field in a JSON structure or a unique value, you can query directly into the structure to that level and use LOOP to find information, and even use the result to traverse a structure backwards. If you know 'c' is a unique field value:

```
outputmacro: ^findarray($_value)
    @0 = ^query(direct_o ? ? $_value)
    ^LOOP()
    {
        $_f = ^first(@0fact)
        $_subject = ^field($_f subject)
        if (!^jsonkind($_subject)) {^next(LOOP)}
        ^return ($_subject)
    }
    ^fail(CALL)
```

and you can similarly do a search for a unique field name to get its value:

```
outputmacro: ^findarray($_field)
    @0 = ^query(direct_v ?  $_valu$_field ?)
    ^LOOP()
    {
        $_f = ^first(@0fact)
        $_subject = ^field($_f subject)
        if (^jsonkind($_subject))
        {
            $_value =  ^field($f object )
            ^return ($_value)
        }
    }
    ^fail(CALL)
```