ChatScript Pattern Redux

Revision 3/4/2017 cs7.3

Pattern matching information was introduced in the Beginner manual and expanded in the Advanced Manual. Since pattern matching is of such importance, this concise manual lists everything about patterns in one place. Note, despite the extraordinary range of weird matching abilities, almost all of my normal code is based on one of three patterns:

```
# rule 1
u: (![plastic] << bag trick >>)

# rule 2
u: ( I * love * you )

# rule 3
t: What fruit do you like?
    a: (~why)
    a: (orange)
    a: (apple)
    a: (~vegetables)
```

Rule 1 - searches for key words in any order. While there is a normal order to questions, e.g., *where do you live*, one can ask *you live where?* so handling arbitrary order is generally valuable. Just have all the keywords you need to detect a meaning and use `![...]` to get rid of interpretations you don't want.

Rule 2 - requires an order when both first person and second person pronouns are involved, since order will matter.

Rule 3 - uses simple keywords or concept sets in rejoinders, since the context of the gambit constrains the input so highly.

IF Patterns

Pattern matching can be done not just in a rule's pattern component but also in its output component, within an IF statement as follows:

```
if (pattern _~conjunction ... ) { }
```

That is, if the first word in the test condition is the word pattern, the rest is treated as a standard pattern of a rule (not using AND OR etc). You can capture data here or do anything a normal pattern does.

Pattern Position

A pattern consists of tokens. By default, any normal word in canonical form can match any form of the word, so *he* in a pattern can match *him*, *he*, *his*. A pattern aborts when a token fails to match unless allowed to not match.

The performance cost of a pattern generally is linear in the number of tokens processed. That means these two rules take the same time to match (other than the imperceptable time difference to read the longer token).

```
u: (apple)
u: (~ten_thousand_names_of_fruits)
```

The system tracks current position in the sentence as it matches.The first token of a pattern is allowed to match anywhere in the sentence. After that normally tokens are matched in against words in consecutive order in the input. If a pattern starts to match and then fails, the system is allowed to retry matching later in the sentence once. It does this by freeing up the first matching word/concept token and letting it rebind later. Given this rule:

```
u: ( I like apple)
```

The input *Do you know that I like oranges and I like apples* would match as follows. The first pattern token `I` would match the first *I* partway into the sentence (because it is allowed to match anywhere. The next pattern token `like` is required to match the next input word in the sentence, which it does. The third pattern word *apple* fails to match *oranges*. We just failed. But we have one retry left. So *I* is sought deeper in the sentence and matched. `like` matches *like* and `apple` matches *apples*. So we match. Had that not matched, no more retries exist so the failure sticks. There are tokens you can use that alter the rules/location around current position.

Pattern Constituents

Type of Sentence - `s: ?:`

A responder beginning with `s:` or `?:` implictly is testing that the sentence is a statement or a question. It is built in even before the pattern. All other rules are not immediately sensitive to kind of sentence.

Existence - word `~concept $var %sysvar _0 @0 ^var ? ~`

Basic pattern matching is against words or concepts. Does this word or concept exist?

```
u: ( this ~animal )
```

matches *this dog* or *this dogs* but not *this is my dog*

You can also ask if a user variable is defined just by naming it:

```
u: ( $myvar help)
```

this only matches if input has *help* and `$myvar` is not null.

System variables one would not ask if they are defined (they almost always are) but would use in a relation instead.

Similarly, `_0` by itself in a pattern means is it defined, that is, not null.

```
u: ( _{apple orange} _0 )
```

matches only if apple or orange got matched. And `@0` by itself means does this fact-set have any facts stored in it.

You can also reference an argument to a function call, and its value will be used to decide what to do. A stand-alone `?` means is this sentence a question.

`!?` would test if it is not a question.

A stand-alone `~` means the current topic is already on the pending topic list (was recently considered an active topic).

Grouping Pairs - `()` `[]` `{}` `<<` `>>`

`( ... )` - Parens mean the tokens within must be found "in sequence". The notation of a pattern starts with parens, but has the unusual property of allowing the match to occur anywhere within the sentence, not just at the start. Any nested parens do not have that property, and still require in sequence.

```
u: ( this (is my) pattern)
```

matches *this is my pattern* and not *this sometimes is my pattern*

`[ ... ]` - Brackets mean match one of contained tokens, in the order given. A bracket list tries all its members in sequence, stopping when it finds a match. For the input *I go home for Christmas* this will not match:

```
u: ( [~noun ~verb] * home )
```

because `~noun` will match to *home* and then *home* cannot be found later. On a retry, `~noun` will match to *Christmas*. Since `~noun` can match multiple times, `~verb` never gets tried.

You can composite things like:

```
u: ( [ apple pear (favorite fruit) cherry ] )
```

to match *I eat pear* and *my favorite fruit*, but this form is unlikely to be used in normal CS.

Note that `[ ... ]` and `~concept` are similar but different in important ways. Matching `~concept` is faster than the corresponding list inside `[]` because naming the concept only requires one token. But it takes more memory to store the concept than it does to put the words inside the `[]`.

The other fundamental difference is in position. Words in `[]` are matched in the order given, possibly moving your position mark deep into the sentence. Words in a concept are all matched simultaneously, so which one is found first in the sentence is what sets the position. For an input *I like beer but not wine*

```
u: (I like * ~drinks)
```

this would match beer if beer and wine are in that concept in any order.

```
u: (I like * [wine beer])
```

this would match wine even though it is farther in the sentence.

`<< ... >>` - angles mean match all of the contained tokens in any order. Putting `*` in this kind of pattern is illegal because it has no meaning. Position is not relevant anyway. Position is freely reset to the start following this sequence so if you had the pattern:

```
u: ( I * like << really >> photos)
```

and input *photos I really like* then it would match because it found *I * like* then found anywhere really and then reset the position freely back to start and found photos anywhere in the sentence.

`{ ... }` - Braces means match one of the contained tokens if you can, but don't fail if you don't. Using `{}` inside of angles is pointless (unless you put an underscore in front to memorize something) because it makes no difference to matching whether or not you had the `{}` content. It is not helping align position within the sentence. These are normally used to assist in positional alignment by swallowing words.

```
u: ( I go to {the} market)
```

matches both *I go to the market* and *I go to market*. If you use underscore before braces to memorize the answer found, then when no answer is found the match variable is set to null (no content) but it is set.

Wildcards - * *~2 *3 *-2

Wildcards allow you to relax the positional requirements for matching. The classic wildcard "*" allows you to have zero or more words between other tokens in a pattern.

```
u: ( I * you )
```

matches _I love chicken and hate you- as well as *I you they*.

You can limit the unlimited range by adding `~n` after it. So `*~1` means 0 or 1 words may intervene.

`*~2` is what I commonly use to restrict a range. This allows a determiner and an adjective to fit before a noun, for example, but not allow a pattern to match weirdly.

```
u: ( I like *~2 cat)
```

matches *I like my cats* or *I like a yellow cat.* You can also request a match of a specific number of words in succession using `*n`. `*1` means get the next word. If you are already positionally on the end of the sentence, this match fails. If you aren't sure how many words are left, you could do something like this:

```
u: ( apple _[*4 *3 *2 *1] )
```

which will grab the next 4 or 3 or 2 or 1 words, depending on how many are available. Generally done with an underscore in front to memorize the sequence.

`*-2` is like `*2`, only it matchs backwards instead of forwards. Valid thru `*-9`.

Negation `!` and `!!`

`!x` means prove that x is not found anywhere in the sentence later than where we are:

```
u: (!not I love you)
```

This pattern says the word not cannot occur anywhere in the sentence. `!!x` means prove that x is not the next word.

Original Form - `'`

While CS normally matches both original and canonical forms of words when you give a pattern word in canonical form, you can require it only match the original form by quoting it. `u: (I * 'take it)` does not match *I am taking it*

Likewise in a relation where you use a match variable, quoting it means use only its original value. `u: (_~fruits '_0==apple)` matches *I like apple* but not *I like apples*

Literal Next - `\`

You can tell CS that a token should be considered a token, not a special form, by putting a `\` in front of it. This applies to single characters like: `\[ \]` and it also applies to relational tokens like `\tom=*` which means do not treat this as a relational test, but instead as a token whose name is wildcarded. Note that

the \ does not suppress detecting the * in a word and therefore allowing variant spelling.

Composite Words - "xxx"

There are sequences of words that have a specific meaning and are treated as a single word, e.g., batting cage. In a dictionary these are often represented using an _ instead of a space, e.g., `batting_cage`. When CS tokenized your input, it automatically converts your separated input words into ones with underscores in them when appropriate. They are no longer single words, but instead a single composite word. This would normally mean that

```
u: (batting cage)
```

would not match. But the script compiler does the same tokenization thing, so your actual internal pattern looks like:

```
u: (batting_cage)
```

For clarity, it is recommended that when you know you are dealing with a composite word, you use the underscore notation.

Sequences of words can also be designated using double quotes.

```
u: ("batting cage")
```

CS converts a quoted string into the same underscore notation. The distinction between the two is generally one of documentation. I use quoted strings for phrases to highlight the intention that they are a phrase. I also use them for multiple word proper names like *Eiffel Tower*. It is particularly important to use the quoted notation when punctuation is embedded in the name like *John's Apple Pie* because knowing where to put underscores when punctuation is involved is tricky. By using quotes, you tell the system to manage things appropriately (`John_'s_Apple_Pie`)

When using the quoted notation, the system will actually try to match original and canonical, just like with ordinary words. If all words in phrase are canonical, the system will match any form of each word. If one is not canonical, it can only match the original form.

```
u: ("king of the jungle")
```

cannot match *kings of the jungle* because *the* in pattern is not canonical.

```
u: ("king of a jungle")
```

but the above rule can match *kings of the jungle* since all words in the quote are canonical.

Memorization - _

Placing an underscore means to memorize what was matched onto a match variable. Match variables are allocated in sequence in a pattern, starting with `_0` and increasing to `_1` etc for each memorized match. The system memorizes the original word, the canonical word, and the position in the sentence of the match.

Relations - > < ? == != <= >=

You can test relationships by conjoining a token with a relationship operator and another token, with no spaces. E.g.,

```
u: (I am _~number > _0>18) You are of legal age.
```

The relationship operators are: `==` - equal

`!=` - not equal

`<` - less than

`<=` - less than or equal to

`>` - greater than

`>=` - greater than or equal to

`&` - bit anded results in non-zero

`?` - is member of 2nd arg concept or topic

`?` - if no argument occurs after, means is value found in sentence

Comparing two text strings (not numbers) will do it based on case-independent alpha sorting.

Note: You can put `!` before the tokens instead of using `!=` and `!?`. E.g.,

```
u: ( _~noun !_0?~fruit) if the noun is not in fruit concept
```

The stand-alone `?` is used with variables for dynamic matching. While you cannot do memorization in front of a comparison (because no positional data is gained) you can in front of the ? operator since finding where in the sentence something is will return a position for memorization.

`u: ( $var? )` means is the value of `$var` found in the sentence anywhere

Note that when `$var` is a normal word, that is simple for CS to handle. If `$var` is a phrase, then generally CS cannot match it. This is because for phrases, CS needs to know in advance that a phrase can be matched. If you put *take a seat* as a keyword in a concept or topic or pattern, that phrase is stored in the dictionary and marked as a pattern phrase, meaning if the phrase is ever seen in a sentence, it should be noticed and marked so it can be matched in a pattern.

But if it is merely in a variable, then the dictionary is unaware of the phrase and so `$var?` will not work for it.

There is also a ?$var form, which means see if the value of the variable is findable. The value can be either a word or a concept name.

Escape - \

If you want to match a reserved punctuation symbol like `[` or `(`, you must escape it by putting a backslash in front. This is commonly done in matching out-of-band information.

```
u: (< \[ * \] ) ^respond(~determine_oob)
```

One also uses escape if you want to know if the sentence was punctuated with an exclamation.

```
s: ( \! )
```

means user did something like *I love you!*.

You may use either `?` or `\?` when asking if the sentence has a question in it. You'd generally only do this in a rejoinder.

Function Call - ^xxx(...)

You can call a function from within a pattern. If the function returns a failure code of any kind, the match fails. If the function is a predefined system function, you are allowed relation operators on the result as well.

```
u: ( ^lastused(~ gambit)>5 )
```

User defined functions (`patternmacros`) do not allow relational operators after them. Patternmacros do not generate answers. They are treated as in-line additional pattern tokens.

```
Patternmacro: ^testuse(^value) _~noun _0==^value
u: ( ~noun ^testuse(apple)) matches "I like pear and apple"
```

A powerful use of function calling is to call `^respond(~topicname)` from in a pattern. The topic can match something and set up a variable for further guidance. E.g.,

```
u: (^respond(~finddelay) $$delay) Wait for $$delay.
```

`~Finddelay` can hunt for time referred to in seconds, minutes, hours, etc, or in words like next week or tomorrow or whatever complex matching you want to do.

Partially Spelled words: `*ing bottle* 8bott*`

You can request a match against a partial spelling of a word in various ways. If you use `*` somewhere after an alpha, it matches any number of characters.

`u: (sag*us)` matches many misspellings of *sagittarius*

If you use `*` followed by an alpha, you get anything as a prefix followed by what you request.

`u: ( *tha)` matches *Martha*

If you put a number in front, it means the word must be exactly that many characters long, matching your pattern. `u: ( 6sit*)` matches *sitter*

When using an `*` word, you can use `.` to indicate exactly one character of any value.

`u: ( sit*u.tion)` matches *situation*

Altering Position - `< > @_0+ @_0- @_0`

When you put `<` in your pattern, it doesn't actually match anything. It means "reset position" to the start of the sentence.

```
u: ( < I love )
```

matches *I love* but not *do I love*

When you put `>` in your pattern, it does not alter your position, but it tries to match to confirm you are on the last word of the sentence.

```
u: (I * > )
```

in this pattern `>` is redundant, since `*` would match to the end of the sentence anyway. You may also use `!>` to ask that we NOT be at the end of the sentence.

`@_1+` says to set the position to where the given match variable (`_1`) matched. Positional sequencing will continue normally increasing thereafter. You can suffix the match variable with `-` instead, to tell CS to begin matching in reverse order in the sentence, i.e., matching backwards to the start of the sentence.

When you use `+`, the position starts at the end of the match. When you use `-`, the position starts at the start of the match.

```
u: ( _home is @_0- pretty)
```

matches *my pretty home is near here.*

Note when you use `-` for reverse matching, the behavior of `<` and `>` changes. `>` sets a position and `<` confirms it instead of the way it is for `+`.

When you omit either + or -, you create a matchable anchor like `@_0`. It represents what was found at that position, and during the pattern must also match at that location now.

```
u: ( _@0 is @_1)
```

The above pattern says that the word `is` must be precisely found between the locations referenced by @0 and @1.

Debugging

`:testpattern`

The system inputs the sentence and tests the pattern you provide against it. It tells you whether it matched or failed.

```
:testpattern ( it died ) Do you know if it died?
```

Some patterns require variables to be set up certain ways. You can perform assignments prior to the sentence.

```
:testpattern ($gender=male hit) $gender = male hit me
```

Typically you might use `:testpattern` to see if a subset of your pattern that fails works, trying to identify what has gone wrong. You can also name an existing rule, rather than supply a pattern.

```
:testpattern ~aliens.ufo do you believe in aliens?
```

`:prepare`

Since CS may revise your input for various reasons, to know why a pattern fails you may need to know what actually say. Using `:prepare` will tell you what the final input words were, and what concepts got marked.

```
:prepare This is a sentence.
```

`:verify`

In general all of your responders and rejoinders should have a sample input comment above them.

```
#! Do you believe in dogs?
?: ( << you believe dog >>) I do.
```

This allows you to do

```
:verify ~mytopic pattern
```

and have the system test if your rule would match your input.

```
:trace
```

You can get a trace of various system functions.

```
:trace pattern
```

will show you pattern matching and match variable binding. Also useful if done before `:testpattern`.

Overruling/Supplementing CS Matching

Sometimes you want to supplement the marking of concepts done by adding your own marks. This is particularly useful handling idioms where no keyword exists. I set $cs_prepass to be a topic which looks for idioms.

```
?: ( < what do you do > ) ^mark(~work)
```

This will cause the work topic to react later as though one of its keywords was given. Likewise sometimes you want to disable some marking. For example, *chocolate* is both a flavor and a color. To avoid going to the color topic incorrectly I might do this:

```
u: ( << _chocolate [taste eat] >>) ^unmark(~colorTopic _0)
```

If the above rule detects *chocolate* in the apparent context of eating, it will unmark any reference to `~colortopic` found at the location of the word *chocolate*.

Graduation Exercise

The pattern matching system of ChatScript has esoteric abilities. I was asked if I would implement an additional one that would look something like this:

```
<< green nose mucus >>~3
```

which he wanted to mean: find all those words, in any order, with each word after the first within a gap range of 3 from the previous word. So it could recognize: *green is the nose with my mucus* or *my nose puts forth green mucus* and not match *while green is my favorite color* or *I don't want to see it in my nose mucus*.

I replied it could probably already be done in CS as it stood, and a few minutes later had whipped up code to do that. Your advanced challenge, if you care to think about it and really warp your mind, is to think of a way to do it yourself. That will prove you really understand what can be done in pattern matching. Answer is on the next page.

```
patternmacro: ^nearbyword(^word)
[
(@_0+ *~3 _^word ^eval(_0 = _1))
(@_0- *~3 _^word ^eval(_0 = _1))
]
```

The macro contains two choices, a sequence that looks forward from where you are and a sequence that looks backwards. Using a nested `()` the system will effectively treat that as a single match, which makes it a single token to be used in a `[]` choice. Whichever `()` finds the next word, it memorizes where it is, then sets the current `_0` location to the new word and the choice ends. While you can't do code execution directly in a pattern, and you can't call out to user-defined outputmacros, you can call out to system functions, and `^eval` lets you do any amount of normal code execution. So this allows us to assign the new match variable to the old. And assigning match variables means assigning all of their attributes, including original value, canonical value, and actual position data.

```
topic: ~test()
u: ( _green ^nearbyword(nose) ^nearbyword(mucus)) You have a disgusting nose.
```

The test pattern therefore, finds the first word and sets the current `_0` location. Then it uses the macro to find the next word and change location, and then the next word.