ChatScript Advanced Pattern Manual

ADVANCED PATTERNS

UNLIMITED WILDCARDS

When you use * you want everything matching until the next significant token, eg

```
u: (I love *  tomorrow) matches all words between love and tomorrow.
u: (I love * ) matches all words after love and up to the implied > (end of sentence)
```

and similarly *~5 wants to match up to 5 words betwen here and the next pattern word. But this breaks down if the next token after the wildcard is not a word or the end. CS must resolve the gap involved on the next pattern token. For example you can't do this:

```
u: ( I love _* $var:=_0 )
```

For capturing the rest of sentence you can do this:

```
u: ( I love _* > $var:=_0 )
```

But you can't intrude pattern assignments or function calls or whatever between actual words and concepts, a wildcard, and then more words and concepts.

Keyword Phrases

You cannot make a concept with a member whose string includes starting or trailing blanks, like " X ". Such a word could never match as a pattern, since spaces are skipped over. But you can make it respond to idiomatic phrases and multiple words. Just put them in quotes, e.g.

```
concept: ~remove ( "take away" remove )
```

Normally in patterns you can write

```
?: ( do you take away cheese )
```

and the system will match sentences with those words in order.

In WordNet, some words are actually composite words like : `TV_show`. When you do `:prepare` on *what is your favorite TV show* you will discover that the engine has merged `TV_show` into one composite word. The system has no trouble matching inputs where the words are split apart

```
?: ( what is your favorite TV show )
```

But if you tried a word memorize like

```
?: ( what is your favorite *1 *1 )
```

that would fail because the first `*1` memorizes `TV_show` and there is therefore no second word to memorize.

Likewise when you write

```
concept: ~viewing ("TV show")`
```

the system can match that concept readily also. In fact, whenever you write the quoted keyword phrase, if all its words are canonical, you can match canonical and noncanonincal forms. `"TV show"` matchs `TV shows` as well as `TV show`.

Implied concept Sets

When you make a pattern using [] or {} and it contains words, phrases, and concept sets, the system will make an anonymous concept set out of them. This allows the system to find the soonest match of any of them. Otherwise [] and {} take each element in turn and try to find a match, which may be later in the sentence than a later element in the set would match.

Dictionary Keyword sets

In ChatScript, WordNet ontologies are invoked by naming the word, a `~`, and the index of the meaning you want.

```
concept: ~buildings [ shelter~1 living_accomodations~1 building~3 ]
```

The concept `~buildings` represents 760 general and specific building words found in the WordNet dictionary any word which is a child of: definition 1 of shelter, definition 1 of accommodations, or definition 3 of building in WordNet's ontology.

How would you be able to figure out creating this? This is described under `:up` in Word Commands later.

`Building~3` and `building~3n` are equivalent. Note, however, that CS does not compile your named meaning into the script as is. You are naming a meaning, so CS will find the corresponding master meaning and use that instead (if different). This is because the inheritance hierarchy from below will only come thru the master meaning. For example: if you write prison-break~1, the system will compile break~1 because that is the master, and covers all other specific word meanings that are equivalent including breakout~1.

The first is what you might say to refer to the 3rd meaning of building. Internally `building~3n` denotes the 3rd meaning and its a *noun* meaning.

You may see that in printouts from Chatscript. If you write `building~3n` yourself, the system will strip off the `n` marker as superfluous.

Similarly you can invoke parts of speech classes on words. By default you get all of them. If you write:

```
concept: ~beings [snake mother]
```

then a sentence like *I like mothering my baby* would trigger this concept, as would *He snaked his way through the grass.* But the engine has a dictionary and a part-of-speech tagger, so it often knows what part of speech a word in the sentence is.

You can use that to help prevent false matches to concepts by adding `~n ~v ~a` or `~b` (adverb) after a word.

```
concept: ~beings [snake~n mother~n]
```

If the system isn't sure something is only a noun, it would let the verb match still. Thus a user single-word reply of snakes would be considered both *noun* and *verb.*

The notation `run~46` exists to represent a meaning.

There is mild inherent danger that I might kill off some word meaning that is problematic (eg if `run~23` turned out mark the `~curses` set and I didn't want the resulting confusion), said kill off might strand your meaning by renumbering into either non-existence (in which case the script compiler will warn you) or into a different pos set (because your meaning was on the boundary of meanings of a different pos type).

Use of the specific meaning is handy in defining concepts when the meaning is a noun, because Wordnet has a good noun ontology.

Use of the specific meaning of other parts of speech is more questionable, as Wordnet does not have much ontology for them.

The broader scope meaning restriction by part-of-speech (eg `run~v`) has much more utility. It has its risks in that it depends on the parser getting it right (as you have seen), which over time will get better and better.

In MOST cases, you are better off with the full fledged unadorned word, which is parse-independent.

This is particularly true when you are pattern matching adjacent words and so context is firm. < run `~app` is a pretty clean context which does not need pos-certification.

The topic on `~drugs` would want in its keyword list `clean~a` to allow *I've been clean for months* to target it, but not *I clean my house.*

System Functions

You can call any predefined system function. It will fail the pattern if it returns any fail or end code. It will pass otherwise. The most likely functions you would call would be:

`^query` to see if some fact data could be found. Many functions make no sense to call, because they are only useful on output and their behavior on the pattern side is unpredictable.

Macros

Just as you can use sets to "share" data across rules, you can also write macros to share code.

Pattern macros

A `patternmacro` is a top-level declaration that declares a name, arguments that can be passed, and a set of script to be executed "as though the script code were in place of the original call".

Macro names can be ordinary names or have a `^` in front of them. The arguments must always begin with `^`.

The definition ends with the start of a new top-level declaration or end of file. E.g.

```
patternmacro: ^ISHAIRCOLOR(^who)
    ![not never]
    [
        ( << be ^who [blonde brunette redhead blond ] >> )
        ( << what ^who hair color >> )
    ]

?: ( ^ISHAIRCOLOR(I) ) How would I know your hair color?
```

The above patternmacro takes one argument (who we are talking about). After checking that the sentence is not in the negative, it uses a choice to consider alternative ways of asking what the hair color is.

The first way matches `are you a redhead`. The second way matches `what is my hair color`. The call passes in the value `I` (which will also match my mine etc in the canonical form).

Every place in the macro code where `^who` exists, the actual value passed through will be used.

You cannot omit the ^ prefix in the call. The system has no way to distinguish it otherwise.

Whereas most programming languages 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.

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

When a patternmacro takes a single argument and you want to pass in several, you can wrap them in parens to make them a single argument. Or sometimes brackets. E.g.,

```
?: ( ^DoYouDoThis( (play * baseball) ) ) Yes I do

?: ( ^DoYouDoThis( [swim surf "scuba dive"] ) Yes I do
```

If you call a patternmacro with a string argument, like *"scuba dive"* above, the system will convert that to its internal single-token format just as it would have had it been part of a normal pattern. Quoted strings to output macros are treated differently and left in string form when passed.

You can declare a patternmacro 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.

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

Patterns process a token at a time. A token is characters with no white space (generally). But the system recognizes direct function calls from patterns and the arguments and parens surrounding them may have spaces. But you cannot do assignment statements from a function call. I.e.,

$$tmp :=^{f} oo(a) is not legal. You can get the effect you want with$$

tmp:=^" ^foo(a) " because the active string protects the function call.

Dual macros

You can also declare something dualmacro: which means it can be used in both pattern and output contexts.

A patternmacro cannot be passed a factset name. These are not legal calls:

```
^mymacro(@0)
^mymacro(@0subject)
```

Do not write code in a patternmacro as though it is an output code. You can't do

```
if (...) {}
```

If you want to do that, use an outputmacro and call that from your pattern.

Literal Next \

If you need to test a character that is normally reserved, like `(` or `[`, you can put a backslash in front of it.

```
s: ( \( * \) ) Why are you saying that aside?
```

This also works with entire tokens like:

```
u: ( \test=fort )
```

Normally the above without `\` would be considered a comparison. But the `\` at the start of it says treat `=` as just an ordinary part of the token. You can even put `_` in front of it:

```
u: ( _\test=fort )
```

Note that `\` does not block a word with an `*` in it from performing wildcard spelling.

Question and exclamation - `? !`

Normally you already know that an input was a question because you used the rule type `?:` .

But rejoinders do not have rule types, so if you want to know if something was a question or not, you need to use the `?` keyword. It doesn't change the match position

```
t: Do you like germs?
    a: ( ? ) Why are you asking a question instead of answering me?
    a: ( !? ) I appreciate your statement.
```

If you want to know if an exclamation ended his sentence, just backslash a `!` so it won't be treated as a not request. This doesn't change the match position.

```
s: ( I like \! ) Why so much excitement
```

More comparison tests - `& ?`

You can use the logical and bit-relation to test numbers. Any non-zero value passes.

```
s: ( _~number _0&1 ) Your number is odd.
```

`?` can be used in two ways. As a comparison operator, it allows you to see if the item on the left side is a member of a set (or JSON array) on the right. E.g.

```
u: ( _~propername '_0?~bands )
```

As a standalone, it allows you to ask if a wildcard or variable is in the sentence. E.g.

```
u: ( _1? )
u: ( $bot? )
```

Note that when `_1` is a normal word, that is simple for CS to handle. If `_1` 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 `_1?` will not work for it.

Comparison with C++ #define in `dictionarySystem.h`

You can name a constant from that file as the right hand side of a comparison test by prefixing its name with `#`. E.g.,

```
s: ( _~number _0=#NOUN )
```

Such constants can be done anywhere, not just in a pattern.

Current Topic `~`

Whenever you successfully execute a rule in a topic, that topic becomes a pending topic (if the topic is not declared `system` or `nostay`). When you execute a rule, while the rule is obviously in a topic being tested, it is not necessarily a topic that was in use recently.

You can ask if the system is currently in a topic (meaning was there last volley) via `~`. If the topic is currently on the pending list, then the system will match the `~`, e.g.,

```
u: ( chocolate ~ ) I love chocolate ice cream.
```

The above does not match any use of chocolate, only one where we are already in this topic (like topic: `~ice_cream`) or we were recently in this topic and are reconsidering it now.

A useful idiom is `[~topicname ~]`. This allows you to match if EITHER the user gave keywords of the topic OR you are already in the topic. So:

```
u: ( << chocolate [~ice_cream ~] >> )
```

would match if you only said *chocolate* while inside the topic, or if you said *chocolate ice cream* while outside the topic.

Prefix Wildcard Spelling and Postfix Wildcard Spelling

Some words you just know people will get wrong, like *Schrodinger's cat* or *Sagittarius*.

You can request a partial match by using an * to represent any number of letters (including 0). For example

```
u: ( Sag* ) # This matches "Sagittarius" in a variety of misspellings.

u: ( *tor ) # this matches "reactor".
```

The * can occur in one or more positions and means 0 or more letters match.

A period can be used to match a single letter (but may not start a wildcardprefix). E.g.,

```
u: ( p.t* )
```

can match pituitary or merely pit or pat. You cannot use a wildcard on the first letter of the pattern.

```
u: ( .p* )
```

is not legal because a pattern may not start with a period.

Concept Patterns

Concepts also allow patterns as members, but they are not saved as member facts. They are saved as conceptpattern facts, where the subject is the pattern and the object is the concept name (the verb is conceptpattern). They are executed after the normal NL pipeline is complete. The pattern can be compiled or uncompiled. Coming from compiling script they are compiled. Coming from a function like ^testpattern they may or may not be compiled. If uncompiled, the system will compile them first, every volley. A concept pattern looks like below- i.e. a double-quoted pattern as you would find it in any rule.

```
concept: ~romance( "(boy finds girl)" "( [boy girl] meet [boy girl]) " romance )
```

The above has a concept with 2 concept patterns and a normal word.

Indirect pattern elements

Most patterns are easy to understand because what words they look at is usually staring you in the face.

With indirection, you can pass pattern data from other topics, at a cost of obscurity. Declaring a macro does this. A ^ normally means a macro call (if what follows it is arguments in parens), or a macro argument. The contents of

the macro argument are used in the pattern in its place. Macro arguments only exist inside of macros. But macros don't let you write rules, only pieces of rules.

Normally you use a variable directly. `$$tmp = nil` clears the `$$tmp` variable, for instance, while `u: ( ) $$tmp` will output the value into the output stream. The functional user argument lets you pass pattern data from one topic to another.

```
s: ( are you a _^$var )
```

The contents of `$var` are used at that point in the pattern. Maybe it is a concept set being named. Maybe it's a word. You can also do whole expressions, but if you do you will be at risk because you won't have the script compiler protecting you and properly formatting your data. See also the ChatScript Advanced Variables manual.

Setting Match Position - `@_3+ @_3- @_3`

You can "back up" and continue matching from a prior match position using `@` followed by a match variable previously set. E.g.

```
u: ( _~pronoun * go @_0+ often )
```

This matches *I often go* but not *I go* Just as `<` sets the position pointer to the start, `@_0+` makes the pattern think it just matched that wildcard at that location in the sentence going forward.

```
s: ( _is _~she ) # for input: is mother going this sets _0 to is and _1 to mother
s: ( @_1+ going ) # this completes the match of is mother going
```

OK. Setting positional context is really obscure and probably not for you. So why does it exist? It supports shared code for pseudo parsing.

You can match either forwards or backwards. Normally matching is always done forwards. But you can set the direction of matching at the same time as you set a position. Forward matching is `@_n+` and backward matching is `@_n-`.

Note when using backwards matching, `<` and `>` flip meanings. `>` means start at the end (since you are moving backwards) and `<` means confirm we are at start.

`@_3` is a special positional matching called an anchor. It not only makes the position that given and matching forward thereafter, but it also acts as an item that must itself be matched.

E.g., for this pattern `( @_3 is my @_4 life)`

The position pointer moves to `@_3` because as the opening element it can match anywhere in a sentence, just like a word would. But after it matches (example at word 2) then `is` must be word 3 and `my` must be word 4 and `@_4` must start at word 5 and after it completes then `life` must be the next word. Whereas `(< @_3 is)` implies that `@_3` is at position 1, since `<` says this is sentence start.

Retrying scan `@retry`

Normally one matches a pattern, performs the output code, and if you want to restart the pattern to find the next occurrence of a match, you use ^retry(RULE) or ^retry(TOPRULE). Well, if your pattern executes @retry as a token, it will retry on its own without needing to execute any output code. Useful in conjunction with ^testpattern.

Backward Wildcards

You can request n words before the current position using `*-n`. For example

```
u: ( I love * > _*-1 ) capture last word of sentence
```

Concept intersection keywords

If you join a word (or a concept) and one or more concepts, that represents the intersection of them. e.g., ($_{\text{animals}}$tasty) will reference all animals considered tasty. You may stack no more than 3 together.

```
u: (~animals~tasty~mythical) I didn't know you considered dragons tasty.
```

You can do this in a pattern but not as a keyword of a concept. It only works dynamically during pattern matching.

Note: you cannot use word~1 (meaning specification) or word~n (pos-tag specification) on your first word.

Note: This particular ability has general utiility, but specific utility to German. German doesn't have the same strict word order but because there is case marking on nouns then the position of the subject and object (nominative and accusative case) can move around. The simple << >> grouping is not useful unless one can limit the nouns to the right case, e.g.,

```
<< Mann~noun_nominative Hund~noun_accusative >>
```

Concept intersection is sort of analogous to (_~animals _0?~tasty) but if this pattern detects a non-tasty animal first, it fails. And this is more cumbersome.

```
u: (_~animals)
    if (_0 !? ~tasty) { ^retry(RULE)}
```

Also this only works if the focal memorization if the first thing found in the pattern. Otherwise retry wont be restarting with the right thing to consider.

Assignment in a pattern

Aside from the use of _ to memorize a match, you can directly assign to any variable from any other value using `:=`. You can even do arithmetic for these assignments (:+= :-= “*= :/= :&= and any of the other numeric assignment operators) .

```
$value = 5
( _some_test  $value:=5 $value1:=_0 $value2:='_0 $value3:=%time )
( _some_test  $value:+=5 $value1:-=_0 )
```

This is not obviously useful normally, but is helpful in conjunction with `^testpattern` to return data to a remote CS api user.

If you want to do function call assignment, you can do this:

```
$value:=^"^function(foo d)"
```

The reason you have to do an active string here, is because normally spaces break apart tokens, and a pattern token involving a function needs to have all arguments part of the same token. Hence assigning from an active string, where the double quotes around it prevents the token from breaking apart.

Gory details about strings

‘Strings in Output’

A double quoted string in output retains its quotes.

```
u: () I love "rabbits"
```

will print that out literally, including the double quotes.

And you cannot run the string across multiple lines.

An active string interprets variable references inside. It does not show the containing quotes around the whole thing. And it can be extended across multiple lines (treating line breaks as a single space in the string created).

```
u: ( I "take charge" ) OK.
```

When you use “take charge” it can match taking charges, take charge, etc. When you use *“taking charge”* it can only match that, not *“taking charges”*.

If all words in the string are canonical, it can cover all forms. If any are not, it can only literally match.

The quote notation is typically used in several situations...

- You are matching a recognized phrase so quoting it emphasizes that.

- What you are matching contains embedded punctuation and you don't want to think about how to tokenize it e.g., *"Mrs. Watson's"* – is the period part of Mrs, is the `'` part of *Watson*, etc. Easier just to use quotes and let the system handle it.
- You want to use a phrase inside `[]` or `{}` choices. Like `[ like live "really enjoy" ]`

In actuality, when you quote something, the system generates a correctly tokenized set of words and joins them with underscores into a single word. There is no real difference between `"go back"` and `go_back` in a pattern.

But compared to just listing the words in sequence in your pattern, a quoted expression cannot handle optional choices of words. You can't write `"go {really almost} back"` where that can match `go back` or `go almost back`.

So there is that limitation when using a string inside `[ ]` or `{ }`. But, one can write a pattern for that. While `[ ]` means a choice of words and `{ }` means a choice of optional words, `( )` means these things in sequence. So you could write:

```
u: ( [ next before (go {almost really} back) ] )
```

and that will be a more complex pattern. One almost never has a real use for that capability, but you use `( )` notation all the time, of course. In fact, all rules have an implied `< *` in front of the `()`.

That's what allows them to find a sequence of words starting anywhere in the input. But when you nest `( )` inside, unless you write `< *` yourself, you are committed to remaining in the sequence set up.

As a side note, the quoted expression is faster to match than the `( )` one. That's because the cost of matching is linear in the number of items to test. And a quoted expression (or the `_` equivalent) is a single item, whereas (take charge) is 4 items.

So the first rule below will match faster than the second rule:

```
u: ( "I love you today when" )

u: ( I love you today when )
```

But quoted expressions only work up to 5 words in the expression (one rarely has a need for more) whereas `()` notation takes any number. And using quotes when it isn't a common phrase is obscure and not worth doing.

Generalizing a pattern word

When you want to generalize a word, a handy thing to do is type :concepts word and see what concepts it belongs to, picking a concept that most broadly

expresses your meaning.

The system will show you both concepts and topics that encompass the word. Because topics are more unreliable (contain or may in the future contain words not appropriate to your generalization, topics are always shown as `T~name` rather than the mere `~name`.

The deep view of patterns

You normally think of the pattern matcher as matching words from the sentence. It doesn't. It matches marks. A mark is an arbitrary text string that can be associated with a sentence location. The actual words (but not necessarily the actual case you use) are just marks placed on the actual locations in the sentence where the words exist.

The canonical forms of those words are also such marks. As are the concept set names and topic names which have those words as keywords. And all parser determined pos-tag and parser roles of words.

It gets interesting because marks can also cover a sequential range of locations. That's how the system detects phrases as keywords, idioms from the parser like *I am a little bit crazy* (where *a little bit* is an adverb covering 3 sentence locations) and contiguous phrasal verbs.

And there are functions you can call to set or erase marks on your own (`^mark` and `^unmark`).

Pattern matching involves looking at tokens while having a current sentence location index. Unless you are using a wildcard, your tokens must occur in contiguous order to match. As they match, the current sentence location is updated. But not necessarily updated to the next adjacent location. It will depend on the length of the mark being matched.

So your token might be `~adverb` but that may match a multiple-word sequence, so the location index will be updated to the end of that index.

And you can play with the location index itself, setting it to the location of a previously matched `_` variable and setting whether matching should proceed forwards or backwards through the sentence. Really, the actual capabilities of pattern matching are quite outrageous.

Pattern matching operates token by token. If you have a pattern:

```
u: ( {blue} sky is blue )
```

and you input *the sky is blue*, this pattern will fail, even though the initial `{blue}` is optional.

Optional should not lead a pattern, it is used for word alignment. The first blue is found and so the system locks itself at that point. It then looks to see if the

next word is sky. It's not. Pattern fails.

It then unlocks itself and allows trying to match from the start of the pattern one later. So `{blue}` matches blue at the end of the sentence. It locks itself. The next word is not sky. Pattern fails. There is nothing left to try.

Interesting thing about match variables

Unlike user variables, which are saved with users, match variables (like `_0`) are global to ChatScript.

They are initialized on startup and never destroyed. They are overwritten by a rule that forces a match value onto them and by things like

```
_0 = ^burst(...)
```

or

```
_3 = ^first(@0all)
```

And those may overrun the needed number of variables by 1 to indicate the end of a sequence. But this means typically `_10` and above are easily available as permanent variables that can hold values across all users etc.

This might be handy in a server context and is definitely handy in `:document` mode where user memory can be very transient. Of course remembering what `_10` means as a holding variable is harder, unlike the ones bound to matches from input. So you can use

```
rename: _bettername _12
```

in a script before any uses of `_bettername`, which now mean `_12`.

Also, although whenever you start to execute a rule's pattern the match variables start memorizing at `_0`, you can change that. All you have to do is something like this:

```
u: (^eval(_9 = null)  I love _~meat)
```

The act of setting _9 to a value automatically makes the system set the next variable, so future memorizations start at _10. Equivalently, there is `^setwildcardindex`

```
u: (^setwildcardindex(_10 )  I love _~meat)
```

Precautionary note about `[ ]` and pattern matching retries

When you list a collection of words or concepts within a `[ ]` in a pattern, the system does not try each in turn to find the earliest match. Instead it tries each in turn until it finds a match. Then `[ ]` quits. So if you have a pattern like:

```
u: ( the * [bear raccoon] ate )
```

and an input like *the raccoon ate the bear*, then matching would proceed as

- find the word `the` (position 1 in sentence)
- try to find `bear` starting at position 2 found at position 5
- try to find the word `ate` starting at position 6 fails

The system is allowed to backtrack and see if the first match, for `the` can be made later. So it will try to find the match later than position 1. It would succeed in relocating it to position 4.

It would then try to find the word `bear` afterwards, and succeed at position 5. It would then try to find the word `ate` after that and fail. It would retry trying to reset the pattern to find the match for `the` after position 4 and fail. The match fails.

You can fix this ordering problem by making a concept set of the contents of the `[ ]`, and replacing the `[ ]` with the name of the concept set.

A concept set being matched in a pattern will always find the earliest matching word. The number of elements in a concept set is immaterial both to the order of finding things and to the speed of matching.