ChatScript Practicum: Patterns

‘”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.

THEORY - philosophy and goals

Bots that require the user learn an English command set to function are only slightly better than a GUI. Humans want computers to talk in human language, not visa versa. So our job is to understand as many of the ways a user can say things as possible. This is done using patterns.

Patterns are the lifeblood of ChatScript. The rest of ChatScript may be considered just programming, but patterns blend into art and linguistic skillsets. Pattern syntax is described in the beginners and advanced ChatScript manuals and reprised in the Pattern Redux manual. This manual is not going to cover all of them. It is here to help you make practical decisions about which pattern element to use when.

Patterns, optimization, and the engine

Generally speaking, there is rarely any value in making a pattern choice based on speed or memory considerations. The engine is already highly optimzed for executing rules. My chatbot Rose has around 9,000 responders of 13,500 rules. So she is, in a sense, largely an FAQ bot able to answer questions like “how old is your mother”, “where do you live”, and “what is your favorite fruit.” She averages executing 1500-2000 rules per volley, which takes her around 13 milliseconds. And half or more of that time is spent in the NLP pipeline preparing the input. So optimizing patterns rarely makes sense.

Compilation

The script compiler reads your script, confirms it is legal, and subtley adjusts it for faster execution. Your patterns consist of elements. These include ordinary words (`brain`), concepts (`~animals`), user variables (‘$’), match variables (’_‘), factsets (’@‘), comparisons (`_0>5`), and various special characters like’{‘,’<<‘,’)‘,’!’ and others). The engine expects each token to be uniformly spaced by a separating blank, so however you space your code, the compiler adjusts it appropriately for execution. Because the engine uses the lead character of an element to branch to code handling it, it sometimes adds a prefix code to what you wrote. If you ever look at a trace, you will see them, as well as accelerators.

```
u: MYLABEL ( $test<5) ok. ==> 00y u: 9MYLABEL ( =7$test<5 ) ok.
```

The 00y tells CS how many characters to jump ahead to reach the end of the entire rule. The 9 in front of MYLABEL tells how many characters to skip over to reach the actual pattern. Inside the pattern, because we did a comparison, that is prefixed with `=` and the 7 tells CS how many characters to skip over to find the comparison operator to use.

The time it takes to execute a pattern is roughly a constant times how many pattern elements it has to execute to succeed or fail. There is no difference in time between matching a word (`brain`), matching a phrase of 4 words (`"life is a bowl"`), and matching against a thousand members of a concept (`~animals`). Executing things like `!` and `<` and `{` are only slightly faster because they don’t need to do a dictionary lookup.

Marking

The NLP pipeline takes an incoming sentence from the user and adjusts it in various ways. It will try to fix spelling and capitalization. So if you really want to see what your patterns are trying to match, you should use `:tokenize` to tell you what CS did to the input.

Additionally, CS will mark memberships of words in various concepts, only some of which are shown below.

My	dog	eats	steak.
~pronoun_possessive	~main_subject	~main_verb	~main_object
	~noun_singular	~verb_present	~noun_singular
	~pets	~kindergarten	~food

We have parts of speech, role in sentence, classifcations, age learned, etc. You can match ANY of these in a pattern. Or use them like this:

```
u: (_~main_subject _0?~pets) Do you have a '_0?
```

which finds the main subject and determines if it is a pet animal.

Compared to patterns, marking is slow, consisting of half or more of the entire cpu time. But it means that you can execute lots of patterns very rapidly.

Matching algorithm

What actually happens during pattern matching is that the matcher has two things to look at. It walks your pattern, element by element. Each element is processed in turn, without looking ahead to what the next element is. It either succeeds or fails (or sometimes caches holding data). It also has the sentence, which it walks, keeping track of where it is in the sentence (the position pointer).

It processes the pattern element using the current position and direction in the sentence (while normally it walks forward it can also walk backward). So when it sees a `word`, it looks from the current position and direction to see if it can find your word. If it can't, it fails. If it can, it sees if it meets current constraints on how far from where we are it can be. For a simple `u: (I like worms)` when we process `I` we find it at the earliest moment in the sentence. The pattern is as though it were `u: (< * I like worms)`. Having found `I`, it looks for `like`. If the input were `I really like worms` then it would find `like` at word 3, which is a gap of 2 from the original `I`. The allowed gap is only 1 (words in sequence), so the match fails. Had the pattern been `u: (I *~2 like worms)`, then the legal gap is 2 and the match would succeed. The new position pointer in the sentence is 3 (`like`), moving forwards, and the next pattern element would hunt for `worms`. It would find `worms` at position 4, which is a legal change, so the pattern matches.

While looking at the pattern is always moving forwards through it, the direction of movement through the sentence, which defaults to forwards, can go backwards if `@_n-` is used. Thereafter it moves backwards until changed by `@_n+`.

If the pattern element being processed specifies a wildcard, like `*~2`, the system stores the legal gap and moves on to the next token. It is not legal for that to also be a gap (else the system could not decide how to map the words to the two tokens). Similarly, if your pattern is `u: ( a *~2 {small} *~2 gap)` that would also not be legal (though not detected by the compiler at present), because if `{small}` is not matched, your pattern becomes `u: (a *~2 *~2 gap)`.

Whenever the first real element is found, the system remembers where. If the pattern later fails, it is allowed to unhook that match and try to rematch it later in the sentence. This is the cheap equivalent of exploring all possible match paths, which for efficiency the system does not do. Therefore given an input of `I want snakes and I want cookies` and a pattern of:

```
u: ( I want ~food)
```

the system finds `I` at word 1 and sets the position pointer and the start pointer to 1. Then hunts for `want`, finding it at word 2 legally. And hunts for a `food` concept and finds it at position 7 and that is an illegal gap. It is allowed to

restart the pattern 1 word later than the initial match. So is "unbinds" the starting `I`, and hunts for another, finding it at word 5. Finds `want` at word 6 and a ~food at word 7 and therefore matches.

It CANNOT match input of `I want snakes and want cookies` because it finds word 1 (`I`) and word 2 (`want`) fails `~food` because the gap is wrong. It DOES NOT change to retrying `want` which would actually be found at word 5 and succeed with ~food for word 6. That is a full recovery mechanism like you would find in Prolog but is too expense here. It merely fails instead, because it only uproots the starting match.

Some initial tokens can preclude moving the start of matching for another round. Tokens like `<` and `@_3+` if first in the pattern will mandate the pattern starts in a specific position in the sentence.

Whenever the system sees a wildcard, it can save the gap for 1 round to see what happens. If the next token is a start token (paren, bracket, squiggle), this can be saved across that recursive call to the pattern matcher. Otherwise it must be resolved on the next pattern element.

Patterns == meaning

For all bots, there are two things they have to do. They "understand meaning" and they carry out action. The action part is usually easy. The hard part is understanding meaning. Since bots are not truly intelligent, understanding all meaning is not possible. Instead, bots "hunt" for specific meanings they expect. They do this with patterns.

Overly tight vs overly general

Since bots do not "understand" meaning, it is inevitable you will either underspecify or overspecify their patterns. If you underspecify, the bot is subject to false positives. It thinks something means something but it doesn't.

```
u: (want ~food) I want food too.
```

The above patterns looks for a meaning about wanting food. But it will accept `I dont want meat` as matching. It is too general.

```
u: (I want meat) I want food too.
```

Correspondingly the above pattern is too specific and missing tons of reasonable inputs.

Machine Learning is at an advantage in that it always memorize the exact specific training input and may generalize from that. I say may, because it may not. ML

trained on `I have a '97 Audi` may completely not recognize `I have a 1997 Audi`.

So your task in ChatScript is to strike the appropriate balance between too general and too specific. There will always be sentences the bot screws up as a consequence.

Account for NLP pipeline in your patterns.

Patterns should be written in the correct case. Proper names should be in uppercase, normal words in lower case. Do not try to handle capitalization for the start of a sentence. Likewise when you define a concept, capitalize correctly. This allows CS to echo back the capitalization specified in the concept set when memorizing a use of it, rather than what the user typed. The compile will warn you at times if it thinks your capitalization is wrong.

```
concept: ~carmakes (ford dodge)
#! I have a ford.
u: (_~carmakes) You have a _0. ==>  You have a ford.
```

CS will spit back the wrong casing above. But if you change the contents of the concept set to uppercase names, then it will spit back the correct casing regardless of what the user typed.

The other thing to avoid is writing contractions in your patterns.

```
u: (what's the weather )
```

THIS IS AWFUL! First, because CS normally takes that input and converts it into `what is the weather`, the pattern cannot match. Worse, it adds `what's` into the dictionary as a legal word (foolishly it trusts you). So when the NLP pipeline sees the word `what's`, it leaves it alone as legal, instead of fixing it to the expanded form `what is`. Your pattern will work, but this will break a lot of patterns that were correctly written. You won't believe how many chat sentences begin with `What is...`

Kinds of bots

We can distinuish 3 kinds of bots: FAQ, command and control, and conversational. The patterns needed for these bots can vary a lot.

Command and control bot

The command and control bot maps a user's request for an action to be taken into actually performing that action. It is similar to an FAQ bot in not initiating a

conversation, except that often it needs `entities` (critical pieces of information) to carry out the request. E.g., `What is the weather in Seattle` is a request for weather report (intent) with the entity of location (Seattle). If the user merely said `what is the weather`, the bot might reply with `where?`. But that's the limit of its initiative in the conversation. Once the request is processed, it returns to idle just like the FAQ bot.

A good starting point for building patterns for these bots is to use `Fundamental Meaning`. This is done by taking a sample input and discarding all the words you can that will still allow an average high schooler to understand what is being requested. It's a form of pidgin English. E.g.

```
Please tell me what the weather will be in Seattle tomorrow.
```

You can discard a lot of words. `Please` is merely politeness. `tell me` reduces to just `tell` because the user is talking to the bot so directing the bots output back to the user is not necessary, it is assumed. `tell John` would be different. And if you boil out everything else you can you get

```
tell weather seattle tomorrow
```

If a human pretends they are a weather bot, then that pidgin English should be sufficent to be understood.

Once you have distilled the input to its fundamental meaning, you can then broaden it with synonyms.

```
tell + describe explain discourse speak
weather + rain snow hot cold temperature icy
seattle + (probably has no synonyms)
tomorrow + in/after 1 day, specific date, day of week
```

And other than the imperative verb, you can probably shuffle the order of the rest of the words.

```
tell tomorrow  seattle weather
```

These steps will help you generate patterns that detect a lot of inputs quickly.

ChatScript has an advantage over ML in that it has a dictionary and pre-existing concepts. This is analogous to having hundreds or thousands of training sentences available which don't have to be specified.

FAQ bot

The FAQ bot maps a user's request for information to an answer. Usually the answer is precanned text. The bot sits idle until the request; it answers the question; and then returns to idle.

An FAQ bot needs to determine the intent and usually doesn't care about trying to figure out any entities. With a limited repetoire of questions it can answer, it

may be enough to merely detect relevant keywords. For example, if the FAQ has a question about `what hours are you open`, then a pattern that just listed keywords around that might be sufficient. E.g.,

```
u: ([ hour open available close "what time" lock unlock when ])
    We're open 9-5, 7 days a week.
```

The above covers a lot of ground without being very picky. Of course it might match inappropriately as well. `Are you open to suggestions?`. But people talking to an FAQ bot are generally looking for information, and not as likely to wander sideways.

Conversational bot

A conversational bot is part FAQ bot and part initiator of conversation. If the user asks it a personal question (or any other question), the bot is expected to have an answer (even if it is silly or a quibble) and then the bot should turn around and ask the user something and engage in conversation. If the user is silent for a while, the bot might initiate a gambit to get the conversation flowing again.

Patterns of a conversational bot vary from an FAQ pattern to answer `how old is your mother` to simple keywords to initiate a conversational topic.

```
topic: ~astronomy (sun moon astronomy astronomer star galaxy)
t: We love astronomy. Do you?
```

And off we go into a conversation.

Multiple patterns for a single intent

Since there are multiple ways for a user to express an intent, it follows that you will have to write multiple patterns. One way to do this is in multiple rules.

```
u: TELLNAME (what be you name) My name is Rose.
u: WHATCALL (what be you call) ^reuse(TELLNAME)
```

In this simple example, I use ^reuse in the second rule. This insures that if I change the answer in TELLNAME, it automatically changes for all other rules that ^reuse it. Of course, in this example one could combine the rules into:

```
u: TELLNAME (what be you [name call]) My name is Rose.
```

but that will never hold up to all patterns. For example:

```
u: (who be you) ^reuse(TELLNAME)
```

won't instantly combine. But in fact, I recommend combining them as follows:

```
u: TELLNAME ([
                (what be you [name call])
                (who be you)
            ])
            My name is Rose.
```

This has the advantage of faster execution, smaller code, clear immediate visualization of the patterns and response, least wait in stepping thru code in the debugger. And I recommend each pattern is on a separate line as shown above. I even prefer the response on a separate line from the pattern. It means if you use the debugger to "step in", you see it move to a new line. Clearer.

Clarity in pattterns

A primary rule of thumb for patterns is that they really should be easy to read. For an experienced CS programmer, it should be obvious what you are trying to do. Patterns with nested values of `[]` and `()` and `{}` can be very hard to read. Rather than doing that, it is better to split them into separate patterns. The following is NOT clear.

```
u: ( you *~2 [ ([fine hot foxy sexy] look) (look [fine hot foxy sexy]) ] )
```

and would be clearer if split into two patterns nested inside a pattern:

```
u:  ([
    ( you *~2 [fine hot foxy sexy ] look )
    ( you *~2 look [fine hot foxy sexy ] )
    ])
```

So let's assume that generally you will write your patterns as collections of patterns, and not as ^reuse() calls.

Additionally, you should avoid wrapping pattern sections onto new lines.

```
u:  ([
    ( you *~2 [fine hot
        foxy sexy ]
        look )
    ( you *~2 look [fine hot
        foxy sexy ] )
    ])
```

The above are harder to read/understand when run onto multiple lines.

Furthermore, for every pattern line, you should supply a sample input intended for it. This makes it clearer to a human reader what you are trying to do AND the CS `:verify` command to test your patterns to see if they work. A form of unit test.

```
#! You have a foxy look.
#! You look sexy.
u:   ([
    ( you *~2 [fine hot foxy sexy ] look )
    ( you *~2 look [fine hot foxy sexy ] )
    ])
```

PRACTICE - Specific pattern elements

* vs *~n

Most patterns will work when the user provides only a small amount of input. When they type in paragraph-long sentences, using the unrestricted wildcard `*` has a much higher false detection rate.

```
u: ( I * ~like * you) I like you too.
```

The above rule works fine on input like `I like you`, but is silly if the input is `I like meat but I really loathe you`. The problem can be alleviated by using shorter range wildcards. My favorites are `*~2` and `*~3`. `*~2` is good for skipping noun descriptors. It allows for a determiner and an adjective or an adverb and an adjective.

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

And `*~3` is good for close control of word use:

```
u: (I *~3 ~like *~3 you)
```

You don't expect many words to arise between the subject `I` and the verb `~like`. Nor between the verb and its object.

<< >> vs *~nb

Just as there are problems with the unrestricted wildcard `*`, the any-order construct `<< xxx yyy zzz >>` has similar issues. Of course it does, because it acts like this `xxx < * yyy < * zzz`. One solution is to use the restricted range bidirectional wildcard. This is effective when searching around a particular word, looking for something close before or close after. It avoids having to write two patterns to do the job.

```
u: ( bank *~3b off-shore)   # safe
u: ( << bank off-shore >>)  # unsafe
```

Both can match `I have an off-shore bank account` as well as `my bank account is off-shore`. But the unsafe one matches `We launched our kayak from the banks of the ocean and ended up far off-shore`.

Concepts vs `[ xxx yyy]`

A concept is a list of words or phrases. So is `[ ]` in the middle of a pattern. Does it matter which you use? Absolutely. On a single use basis, the concept takes more memory (irrelevant) but is faster to match. A `[ ]` walks the list, trying to match each word in order. A concept matches all at once as a single operation. It doesn't matter if the concept consists of thousands of members, it matches as fast as a single word. Furthermore, the concept will match the earliest occurence of any word. `[]` will match in the order of words found in the list. So if an earlier word is found late in the sentence, too bad. Given input: `I like your soul next to my shoe`, this pattern:

```
u: (I * [ my your])
```

will match to `my`, even though your is earlier in the sentence. This works out if the pattern is:

```
u: (I *~2 [ my your])
```

because the limitation of *~2 will cause `my` to be found too late, so it will be rejected and the next choice `your` will work.

But it's easy to become confused about what matches when you use `[]`. So concepts are nominally better, clearer. And when used more than once, you only have to edit the concept, not multiple rules. Odds are when you change `[]` in a rule, you will forget to edit other places using the same words.

The problem with using concepts is a) you move the code non-local to the rule so it is no longer obvious what the matching criteria is and 2) you have to create names for your concepts. So while concepts are "better", they can be more tedious to use.

This behavior of `[]` is an efficiency measure. The engine does not do a full search of all possible paths (like Prolog would do) because that is too expensive and rarely pays off. It at best only looks ahead to the next token in the pattern. This allows it to suspend a wildcard for a brief moment.

There are lots of consequences of looking for words in `[ ]` in order. One is that you should put your rarest, most significant words first. Another is that if you really want to find the earliest occurence of the collection, make a concept out of them and search for the concept instead. That is guaranteed to find the earliest occurence.

"xxx yyy" vs (xxx yyy)

Both forms mean a contiguous sequence of words must match. But there are differences. For the quoted form, it matches all words in a single element. So it is faster than all elements within a (). At the top level, separate words are clearer than a quoted phrase.

```
u: (I like you)     # clearer
u: ("I like you")   # slightly less clear
```

Whenever you add a nesting level, patterns are harder to understand. So at nested levels, quoted forms are easier to read than ones in ().

```
u: ([ "I like you" testing])    # clearer
u: ([ ( I like you) testing]    # less clear
```

Things that mitigate against using quoted phrase are the following. 1) You are limited to 5 words in a quoted phrase. 2) A quoted phrase can only match the literal user input or the entirely canonical form.

If the user input is "I loved toys", you can write a quoted phrase for "I love toy" (all canonical) or "I loved toys" (exactly given). You cannot write a pattern like this:

```
u: ("I love toy")       # matchable
u: ("I loved toys")     # matchable
u: ("I love toys")      # unmatchable
```

because the system only detects the original user input OR the canonical form of the input. It does NOT detect a mixture of original and canonical.

When you use individual words, you can select for each whether to be canonical or not because you can put `'` in front of each.

When your phrase incorporates only canonical forms of its words, it can match any form of all of those words. When your phrase has some non-canonical words or it is quoted with `'`, it can only match what the user actually typed (original input).

Using !

Put all ! in front of your pattern, because you avoid wasted effort on the rest

```
#! do you like Vienna
#! do you hate Earth
u: (
    ! [travel movement]
    [
        (you like [Earth Vienna])
        (you hate [Earth Vienna])
```

```
])
```

Using << >>

Similarly, for << >> put the rare stuff first, so if match fails it ends pattern soonest.

{xxx yyy} and [ ] or << >>

{} means optionally find one of these words. It is handy when you are trying to align your pattern to the position pointer in a sentence. It is, however, completely meaningless if nested inside `[ ]` or `<< >>`.

```
u: ANYORDER(  << find {testing available} green)
u: ANYONE(  [ find {testing available} green ])
```

In ANYORDER, since finding words in `{}` is optional, it matters not whether they are found or not. So why include them?

In ANYONE, you are already trying to find one of the words in `[ ]`. So saying that you can use the words in `{}` adds nothing over merely writing

```
u: ANYONE(  [ find testing available green ])
```

<< >> vs < *

Some patterns depend on finding multiple things in any order. The usual pattern for this is `<< xxx yyy zzz >>`. But sometimes CS imperfectly cannot handle certain patterns this way. For example CS 8.0 does not allow !<< xxx yyy >>, even though it should. But there is a workaround. You can do `( xxx < * yyy < * zzz)` to achieve the same effect. That is, find an element anywhere, go to start, find another element anywhere, go to start, find another element anywhere.

@_n+ and @_n- for local context

A lot of times when I find a basic match, I want to subject it to various constraints. It is faster and clearer to find the basic essential keyword, and then look around it for context. Using @_n+ and @_- allows you to jump to where the primary match occured, and then check the local context either testing forwards or testing backwards.

```
u: (_baby) ^refine()
    a: (@_0+ [carriage formula]) # not about a baby, ignore
    a: (@_0- maybe) # title of a movie, ignore
    a: () # We have detected a real baby
```

^setindex() vs _10 = _0

When you match something and check the local context around it using ^refine(), a problem arises when you want to match another piece of data during refinement. Each rule starts memorizing at _0, and you risk clobbering what you have already memorized.

```
u: (_~food) ^refine()
    a: (@_0- _~number)
```

If you are looking for a count before the matched food, you destroy your food as you memorize the number. Two solutions exist. One is to copy the original _0 to a different variable.

```
u: (_~food) _10 = _0 ^refine()
    a: (@_0- _~number)
```

The other is to alter the starting memorization index in your pattern.

```
u: (_~food) ^refine()
    a: (^setwildcardindex(_1) @_0- _~number)
```

Of the two choices, I generally prefer the `_10 = _0` approach, because I do it at the outermost level rule, so dont have to repeat it on multiple rejoinder rules, it takes less typing, and I consider `_0` to be a highly volatile variable that any function I call might destroy also.

Adding markings - ^mark()

The normal engine preparation on your sentence is to mark all words (and their canonical forms) with what concepts they belong in.

You can supplement these marks any time you want. For example:

```
u: (_you) if (^original(_0) == u) {^mark(u _0)} ^retry(RULE)
```

The `texting` substitutions file will change `u` in input into `you`. But suppose you want to detect the actual `u`. The above is such a way. It matches the changed form, checks to see if the original was actually a `u` and marks it in that location. Thereafter the following rule will match.

```
u: (u) Found the letter u.
```

But marking does not make the marked value appear at that position in the sentence.

```
u: (_you) if (^original(_0) == u) {^mark(beer _0)} ^retry(RULE)
```

Had we done the above, we cannot expect this to grab the word `beer`.

```
u: (_beer) I found _0.
```

The pattern will match, but the output will be `I found you`.

You can mark using any word, fake word, concept, or fake concept. There is no restriction.

Unmarking words - `^unmark()`

Not only can you add markings but you can also remove them. This is handy when trying to glean data that is context sensitive.

If we want to detect blood as a body part and it is in the concept `~bodypart` we might use a rule like this:

```
u: (_~bodypart) Found _0.
```

But maybe not all occurences of `blood` are valid. If the following rule occurs earlier, it prevents faulty gleaning.

```
u: (_blood pressure) ^unmark(~bodypart _0) ^retry(RULE)
```

There are actually 2 ways you can unmark. You can unmark a specific word or concept, or you can unmark the entire word. Unmarking the entire word using `*` means CS acts as though it is not in the sentence at all. Given input `I have low blood pressure`, the following is what happens:

```
u: (_blood pressure) ^unmark(* _0) ^retry(RULE)
u: (_*) '_0   -- this prints out `I have low pressure.`
```

Removing the entire word is perhaps a bit drastic, and you may want to reinstate it later. The simplest way to do that is this sequence:

```
u: (_*) _10 = _0   -- memorize the entire sentence location

u: (_blood pressure) ^unmark(* _0) ^retry(RULE)

u: () ^mark(* _10)  -- refresh all hidden words
```

Replacing words - `^replaceword(word _n)`

You can already mark and unmark words, which is what is used for pattern matching. But the word itself in the sentence is what is retrieved when memoriz-

ing a word. You can change the word itself just by providing the word you want used and the location in the sentence (as a match variable). Replacing a word does not make it visible to pattern matching. It is merely what will be retrieved (for both original and canonical).

This is handy, for example, for making it easy to see what was used to create an interjection. If the mark on a word in ~emogoodbye, Then

```
u: (_~emogoodbye)
   $_tmp = ^original(_0)
   ^replaceword($_tmp _0)
```

will make it so when you do this in later patterns:

```
u: (_~emogoodbye) _0 is now the original text
```

Fixing CS substitutions

^unmark and ^mark can be used to "correct" the behavior of standard CS substitutions that you may not want but are unwilling to remove from CS release files. Just detect the substitution result, check the original, and mark and unmark accordingly. For example, "have a nice day" is substituted into `~emogoodbye`. So you can't normally see it in a pattern as `(have a nice day)`. But ... you can find it anyway.

```
u:  (_~emogoodbye)
    $_tmp = ^original(_0)
    $_tmp1 = ^"have a nice day"
    if ($_tmp == $_tmp1) {}
```

And when CS interjection splitting is on (by default), you sometimes get words split over sentence boundaries, where pattern matching can't see them. You can compensate by setting variables in the earlier sentence. So `no, thanks, I hate it` which is really the same as 'thanks, no, I hate it' or 'no, i hate it, thanks' might look like this in code:

```
u: (~no) $$no = 1
u: ([~emothanks thanks]) $$thanks = 1
u: ($$no $$thanks hate it) Hate it if you want. I don't care.
```

Gleaning sentence chunks

^Unmark() is also useful for gleaning data from paired chunks a sentence. In English one might say `if xxx then yyy` or `begins xxxx ends xxx`. You can mask off sentence fragments to perform special gleaning like this:

```
u: (_* from _* to _*) _10 = _0 _11 = _1  _12 = _2
    ^unmark(* _0)  # hide prior to from
```

```
^unmark(* _2)  # hide after to
^respond(~gleantopic) # go find data in remainder
$data1 = $$tmpdata    # save what we learned

^mark(* _2)           # restore to data
^unmark(* _1)         # hide from data
^respond(~gleantopic) # go find data in remainder
$data2 = $$tmpdata    # save what we learned

^mark(* _0)  # restore prior to from
^mark(* _1)  # restore from data
```

Patterns in IF statements

While all rules can have patterns, you can even use pattern matching inside outputmacros or rule output. You tell the `IF` statement you want to use pattern syntax like this:

```
if (PATTERN  $x<5 _~mainsubject)  {}
```

The reality is that outputmacros (functions) can act like rules and rules can act like functions (but you have to pass arguments as globals).

Placeholder rules

It is sometimes handy to have a common rule used for rejoinders from multiple places, or to hold a common output. This can be done using a pattern that cannot match. The most obvious is:

```
s: MYCOMMON (?) Here is common output.
    a: REJOINDER1(how) I dont know how
    a: REJOINDER2(when) I dont know when
...
u: (some pattern) ^reuse(MYCOMMON) # say what MYCOMMON says
u: (some other pattern) ^setrejoinder(OUTPUT MYCOMMON)
    I have this output, and my rejoinder will be handled by a common place.
```

QUIZ - What's wrong with each of these patterns?

```
#! Where do i live
u: Q1( where do i live ?) On the moon?

#! I think bottle deposits are good for the earth.
u: Q2( [bottle can] (deposits are good)) I recycle.

#! Among the fruit that I like are apples.
u: Q3( << [bananas apples] are fruit  >> ) Fruit are tasty.

#! How much does the apple cost?
#! What is the price of a banana?
u: Q4([
        (how much *~5 cost)
        (what be *~5 price)
        !(price of liberty)
      ])
```

ANSWERS

Q1 uses `I` in lower case. And the rule is overly specific. And why use `?` in the pattern when you could change the rule to `?:`.

Q2 has useless interior `( )` so it is not the clearest. If you thought `( deposits are good)` should have been changed to `" deposits are good"`, you missed the clearest answer. You don't need to use `" "` at the top level Of a pattern.

Q3 detects `bananas are fruit` and `examples of fruit are bananas and pineapple` but not `the banana is a fruit`. To do that you need to make banana singular in the pattern (along with `apple`) and change `are` to the more canonical `be`. Of course limiting us to bananas and apples is ridiculous given that CS has the in-built concept `~fruit`. You should type in your sample word like this: `:prepare banana` and see what existing concepts cover it.

Q4 has a negative inside the `[ ]` alternatives so it will match for almost all inputs. It should be moved first, before the `[`. And probably you could combine

the other elements into a single pattern while generalizing further and still being clear.

“‘ u: Q4 (!(price of liberty) [“how much” “what be”] *~5 [cost price fee])

Summary

Fluency in pattern construction requires a limber mind, able to imagine how to broaden a match while not accepting wrong inputs. But it will enable your bots to seem amazingly human!