

# *ktulu* references

Jean-Louis Villecroze

jl@fizz.org @CocoaGeek

*Preview 44*  
*May 30, 2023*

## Module

The `modCTM` module implements a couple of *elemental* objects that supports the development of *ktulu* as an extension of *fizz*. To be available the module must be loaded in the *runtime*, either with the `load` command or by specifying the module's name when using a JSON solution file. For example, if we wanted to execute a test defined in a *fizz* file `foo.fizz`, we would create a `foo.json` file (in the same folder) with the following content:

```
1 {  
2   "solution" : {  
3     "modules" : ["modCTM"],  
4     "sources" : [  
5       "foo.fizz"  
6     ],  
7     "globals" : []  
8   }  
9 }
```

We can then start *fizz* with the JSON file as a command line argument, and it will load the module as well as the *fizz* file (and any other specified file):

```
jl@korok:~/Code/okb/apps/ktulu$ ./fizz.x64 ./etc/ktulu/tests/foo.json  
fizz 0.7.1-X (20200814.0954) [lnx.x64|8|1]  
Press the ESC key at anytime for input prompt  
  
load : loading ./etc/ktulu/tests/foo.json ...  
load : loaded ./mod/lnx/x64/modCTM.so in 0.000s  
load : loading ./etc/ktulu/tests/foo.fizz ...  
load : loaded ./etc/ktulu/tests/foo.fizz in 0.014s  
load : loading completed in 0.015s
```

## *Elemental* classes

The implementation of the *Conscious Turing Machine* as proposed by Manuel, Lenore and Avrim Blum in their article, is built around three core *elementals*: `CTMCBFSActor`, `CTMCFUNActor` and `CTMCStage`.

Unlike in the article, the implementation differentiate between *chunks* by having four possible type of *chunks* for practical reasons:

<b>tweet</b>	a declaration from a <i>processor</i> .
<b>query</b>	a question from a <i>processor</i> in need of replies.
<b>reply</b>	a reply from a <i>processor</i> to another <i>processor</i> 's <i>query</i> .
<b>kudos</b>	a feedback to a <i>processor</i> 's <i>reply</i> to a <i>query</i> .

A *chunk* in this implementation also differs slightly from the article. It is represented as a *list term* with an arity of 8 and contains:

[*originator,tick,unique id,type,gist,weight,intensity,mood*]

The *tick* of a chunk is the CTM's *tick* value for which a given *chunk* was first submitted. If the *chunk* get re-submitted, this value will not be updated.

## CTMCBFSActor

This *elemental* implements the part of a *processor* that communicate with the *stage*: receives STM broadcast and supplies *chunks* for competition. The class is derived from the `MRKCBFSolver` and as such can handle logical queries.

It supports the following *properties*:

<b>stage</b>	a <i>symbol</i> that indicate the <i>CTM</i> to which the actor is connected to. the default is <code>ctm</code> .
<b>dreco</b>	the DROP operator recovery time (in ticks), default is 0 (the actor won't respect the operator).
<b>ctime</b>	how long (in seconds) to keep track of chunks sent and received. If no custom <code>t1</code> value is given to the <i>elemental</i> , a custom value will be set based on <code>ctime</code> and the stage's tick.
<b>wtmod</b>	initial value of the weight modifier based on feedback.
<b>tlink</b>	kudos threshold for linking (default is 5).
<b>links</b>	set to <code>yes</code> to allow the actor to establish direct link with other actors. If links are already established, set to a <i>list</i> that contains <i>lists</i> , each providing: the <i>functor</i> to be used to match patterns, the label of the processors and the last linking value.
<b>rttime</b>	how long (in seconds) to keep track of the content of the STM for. Default is 0.

The following callback *queries* will be sent to the *elemental* in specific situations:

<b>(init)</b>	queried when the <i>elemental</i> is attached to the <i>runtime</i> environment. This only happens once.
<b>(beat,:t)</b>	queried regularly based on the runtime setting. The second <i>term</i> is the last received tick.
<b>(tick,:t,:l,:a)</b>	queried for every broadcast from the <i>stage</i> . The second <i>term</i> is the tick number, the third is the content of the STM (as a <i>list</i> ) and the fourth. is a <i>frame</i> holding ancillary data.
<b>(link,:t,:l)</b>	queried for every query sent directly by another processor. The second <i>term</i> is the tick number at the time of reception and the third is the chunk (in a <i>list</i> ).

The following *logical queries* are available to support posting chunks (they can be called from within the *elemental* it-self or from another):

<b>(tweet,:f,[:g,:w,:m])</b>	submit a <i>tweet</i> like <i>chunk</i> for competition and broadcast to all <i>processors</i> .
<b>(query,:f,[:g,:w,:m])</b>	submit a <i>query</i> like <i>chunk</i> for competition and broadcast to all <i>processors</i> without waiting for any reply.
<b>(query,:f,[:g,:w,:m],:r)</b>	submit a <i>query</i> like <i>chunk</i> for competition and broadcast to all <i>processors</i> .
<b>(reply(:u),:f,[:g,:w,:m])</b>	submit a <i>reply</i> like <i>chunk</i> for competition and broadcast to all <i>processors</i> .
<b>(kudos(:u),:f,[:g,:w,:m])</b>	submit a <i>kudos</i> like <i>chunk</i> for competition and broadcast to all <i>processors</i> .

with `:f` being a standing for a *frame* containing some options for the posting:

<b>retries</b>	the number of ticks for which the <i>actor</i> will automatically post the exact same <i>chunk</i> , until it is included in the STM. If the value is 0 (the default), there won't be any retry. If the value is -1 the actor will retry until the <i>chunk</i> is in the STM.
<b>replace</b>	if set to the value <b>yes</b> , this chunk will replace any previously and still tried posting. (only valid for <i>tweet</i> and <i>query</i> ).
<b>collect</b>	if set to the value <b>yes</b> , the <i>actor</i> will prepend all <i>replies</i> to a <i>list</i> .
<b>diverse</b>	if set to the value <b>yes</b> , the <i>actor</i> will inforce that all <i>replies</i> are different before passing them on.
<b>wide</b>	if set to the value <b>yes</b> , the <i>actor</i> will also submit a query to the competition when it uses a direct link.
<b>repeats</b>	the number of tick for which the <i>actor</i> will automatically post the exact same <i>tweet</i> regardless of getting in the STM. If the value is 0 (the default), there won't be any repost. If the value is -1 the actor will keep posting the <i>tweet</i> until it is replaced.
<b>boost</b>	a factor to be applied to the weight of a chunk when repeating or retrying it. The factor won't be applied when <b>repeats</b> is set to -1.
<b>timeout</b>	when used for a query, this express the number of seconds to expect a reply for. When the timeout occurs, the value <b>null</b> will be returned by the <i>logical query</i> or will be prepended to the <i>list</i> when using the <b>collect</b> option.
<b>dropout</b>	when used for a query, this the amount of weight to take away from any direct links. Ideally, this value should be the same that the one that will be provided with a kudos. Once the weight of a direct link drops bellow a threshold, the link will no longer be used.
<b>kudos</b>	when used for a reply, this a list of tuples (reply uid, fraction) that indicate which replies received in service of answering a query need to be get some fraction of the kudos value received for this reply.
<b>uid</b>	when provided, the value (a <i>symbol</i> ) will be used as the unique id of the chunk to be submitted for competition.

with **:u** being the *unique id* of either the *query chunk* when used to reply, or the *unique id* of the *reply chunk* when used to provide feedback with a *kudos chunks*.

with **:g** being a standing for any *term* that is the *gist* of the *chunk*

with **:w** being the weight of the *chunk* and **:m** being the mood of it.

In a case of a *query*, the *predicate* takes in a fourth *term* which will be unified with any *chunk* received as *reply* (or with a *list* of *chunks* when using **collect**).

To *recall* the content of the STM, as seen by an actor, at some moment in the past, the following *query* can be called upon:

(**recall**, {}, **:filter**, **:result**) Its third *term* is an optional filter on the content of the chunks. Thus to be of use it should be a *list* of arity 8. The fourth *term* will be unified with the *list* of matching *chunks*. The second *term* can be used to specify the option **own** (with the value **yes**) which will restrict the recall to only the chunks of the actor (including the one that failed to appears in the STM).

To *wait* for a specific chunk to be in the STM (or received directly), the following *query* can be called upon:

(await,{},:filter,:result) Its third *term* is an optional filter on the content of the chunks. Thus to be of use it should be a *list* of arity 8. The fourth *term* will be unified with the *list* of matching *chunks*. The second *term* can be used to specify options.

The supported options are:

collect	if set to the value <b>yes</b> , the <i>actor</i> will prepend all <i>chunk</i> to a <i>list</i> .
limit	the maximum number of chunks to be provided by this query.
timeout	the number of seconds to await for (0 for no timeout).

## CTMCFUNActor

This *elemental*, implements a *processor* on top of a FZZCFUNRunner *elemental*. Most of what is described for CTMCBFSAActor is valid for this *elemental* as well, except for the callbacks *queries*. Since this *elemental* is *imperative* based (rather than *logical*), instead of providing *prototypes* to handle the queries, you will need to define *functions*. For example:

```

1 ctm.actor.foo {
2   class      = CTMCFUNActor,
3   primitives = funx.primitive,
4 } {
5
6   (init,[],[ // function to handle the attachment of the elemental to the substrate
7     // your code here
8   ]);
9
10  (beat,[t],[ // function to handle the substrate's pulse
11    // your code here
12  ]);
13
14  (tick,[t,1,a],[ // function to handle CTM's ticks
15    // your code here
16  ]);
17
18 }
```

## CTMCPYTAActor

This *elemental*, implements a *processor* as a *Python* driven *elemental*. Most of what is described for CTMCBFSAActor is valid for this *elemental* as well. It does, however have some *properties* that are specific:

path	the folder in which the python module may be located.
name	the name of the python module to be loaded.
object	a <i>symbol</i> or <i>functor</i> which will be interpreted as the class of object to be instantiated.

For a *Python* class to be usable as a processor, it must have **fizz.Actor** as its parent class. The following methods can be implemented in the derived class to handle the events coming from the processor connection to the CTM:

onInit(self)	called when the processor is first created and attached to the substrate.
onExit(self)	called when the processor is terminated and detached from the substrate.
onBeat(self)	called regularly (at an interval shared by all elementals in the substrate).
onTick(self,tick,chunks,ancillary)	called at each tick of the CTM.
onLink(self,tick,chunks,ancillary)	called when the processor received chunk(s) via a direct link.
onReply(self,uid,chunk)	called when a reply to a query posted by the processor was received.
onKudos(self,uid,chunk)	called when a kudos to a reply the processor sent was received.

The following methods can be used by the processor:

<code>query(options,gist,weight,mood)</code>	post a query out for competition. On return, the UUID of the chunk is provided.
<code>tweet(options,gist,weight,mood)</code>	post a tweet out for competition. On return, the UUID of the chunk is provided.
<code>reply(uid,options,gist,weight,mood)</code>	post a reply ,for a specific query, out for competition. On return, the UUID of the chunk is provided.
<code>kudos(uid,options,gist,weight,mood)</code>	post a kudos ,for a specific reply, out for competition. On return, the UUID of the chunk is provided.

Here's an example:

```
1 import fizz
2 import random
3
4 # define an actor (a.k.a CTM processor) which publish a tweet for competition to the STM
5 class Noise(fizz.Actor):
6
7     def __init__(self,label):
8         self.label = label
9
10    def onInit(self):
11        pass
12
13    def onExit(self):
14        pass
15
16    def onBeat(self):
17        print(self.label + ' : ' + 'beat')
18
19    # at each new tick, we tweet something (80% of the time)
20    def onTick(self,tick,chunks,ancillary):
21        if random.randint(0,100) > 80:
22            self.tweet({},'bazinga.py',1,1)
23
24    def onReply(self,uid,chunk):
25        pass
26
27    def onKudos(self,uid,chunk):
28        pass
```

Since this *elemental* relies on the *Python* extension to *fizz*, the module `modPYT` must be loaded for this *elemental* class to be available. Also, a slightly different version of `modCTM`: `modCTM-PY` must be used.

## CTMCStage

This *elemental* implements the *stage*. Its purpose is to collect all *processor* supplied *chunks*, perform the *up-tree competition* then broadcast the content of its *STM* to all *processors*.

It supports the following *properties*:

<b>mode</b>	the operation mode: <b>probabilistic</b> , <b>deterministic</b> or <b>weightcentric</b>
<b>size</b>	the size of the STM (that is the number of <i>chunks</i> that will get broadcasted at each tick).
<b>ttag</b>	accepted tick lag (in ticks), default is 0. If a <i>chunk</i> is delivered late it will be ignored.
<b>tick</b>	tick frequency (ms) or the label of a ticker <i>elemental</i>
<b>tick.min</b>	when provided, the stage will adjust, as needed, the tick frequency based on the number of submitted chunks between this value and the tick value.
<b>mood</b>	mood factor value.
<b>drop</b>	DROP operator value.
<b>intr</b>	interruption's intensity.
<b>func</b>	competition function.
<b>esel</b>	weight equality selector.

The *competition functions* currently supported are:

<b>abs_weight</b>	use the absolute value of a chunk's weight.
<b>intensity</b>	use the value of a chunk's intensity.
<b>intensity_mood</b>	use the sum of the chunk's intensity with its mood.
<b>abs_mood</b>	use the absolute value of a chunk's mood.
<b>intensity_mood_2</b>	use the sum of the chunk's intensity with its mood halved.

The *weight equality selectors* currently supported are:

<b>left</b>	pick the <i>chunk</i> on the left (of the tree node).
<b>right</b>	pick the <i>chunk</i> on the right (of the tree node).
<b>toss</b>	pick one of the <i>chunk</i> randomly.

At every ticks, the *stage* broadcasts out the content of the STM by publishing a *statement* into the *runtime*. The label of the *statement* is the label of the *elemental* with the postfix `.tick`. The terms in the statement are: the tick number, a *list* with the STM content and a *frame* containing some details on the CTM:

<b>drop</b>	the DROP operator value (only if when deterministic).
<b>intr</b>	interruption's intensity.
<b>chunks</b>	the total number of chunks received by the CTM for this tick.
<b>links</b>	all the chunks that were send directly between processors during that tick.
<b>time</b>	the time of the tick.
<b>mood</b>	mood factor value.
<b>tick</b>	the elapsed tick value.

## A simple example

Let's briefly look at how to setup a very simple test, where two *processors* are competing in a *deterministic CTM*.

First, create a new *fizz* file called `hello.fizz`. In it, we will first provide the definition of the *stage*:

```

1 ctm {
2
3   class    = CTMCStage,
4
5   mode     = deterministic,
6   size     = 1,
7   tick     = 250,
8   drop     = 0.5,
9   func     = abs_weight,
10  esel     = toss
11
12 }
```

The content of the *STM* will get broadcasted 4 times per seconds and the up-tree competition will use the absolute value of each *chunks*' weight. If two *chunks* have the same result for the *chunk function*, the *stage* will randomly pick.

Create now a file `hello.json` to setup the solution:

```
1 {
2   "solution" : {
3     "modules" :   ["modCTM"],
4     "sources" :   [
5       "hello.fizz"
6     ],
7     "globals" :   []
8   }
9 }
```

Before trying this out, let's add another *elemental* definition in `hello.fizz` so that we can see the *STM* content at each broadcast:

```
1 ctm.obs {
2
3   ()  :-  @ctm.tick(:t,:l,_),
4         console.puts(:t," ",:l);
5
6 }
```

Note here, that the *elemental* `ctm.obs` is not considered a *processor*, but takes advantage of the fact that the content of the *STM* is broadcasted as a logical *statement* within the *runtime*, it thus can be used as a *trigger predicate*.

We can now, launch *fizz* with the solution file and observe the content of the STM:

```
jlv@korok:~/Code/okb/apps/ktulu$ ./fizz.x64 ./etc/ktulu/tests/hello.json
fizz 0.7.1-X (20200814.0954) [lnx.x64|8|1]
Press the ESC key at anytime for input prompt

load : loading ./etc/ktulu/tests/hello.json ...
load : loaded ./mod/lnx/x64/modCTM.so in 0.000s
load : loading ./etc/ktulu/tests/hello.fizz ...
load : loaded ./etc/ktulu/tests/hello.fizz in 0.002s
load : loading completed in 0.003s
1 []
2 []
3 []
4 []
5 []
6 []
7 []
8 []
9 []
10 []
11 []
```

Since there is no *processors* yet, the content of the *STM* is empty, as expected. Let's now add to `hello.fizz` a first *processor* which will *tweet* frequently, but not all the time. For this, we will use an *elemental* that generate *statements* every 0.5 seconds:

```
1 ctm.proc.tick {
2   class      = FZZCTicker,
3   tick       = 0.5
4 } {}
```

In the *processor*, we will use `ctm.proc.tick` as a *trigger predicate* to post a *tweet* some of the time:

```
1 ctm.proc.sheldon {
2   class      = CTMCBFSActor,
3   dreco      = 4,
4   weight     = 1,
5   mood       = 1
6 } {
7
8   () :- @ctm.proc.tick(_,_),
9         rnd.real(1,_?[gt(0.3)]),
10        ~self(tweet,{},[bazinga,$weight,$mood]);
11
12 }
```

On line 9, we use the *primitive* `rnd.real` to draw a random number (between 0 and 1) and use it to only *tweet* 70% of the time. On the following line, we query the *elemental* itself to post a *tweet* which *gist* is the *symbol* `bazinga`. The weight and the mood of the *chunk* are taken from the *elemental* properties.

Let's try this:

```
jl@korok:~/Code/okb/apps/ktulu$ ./fizz.x64 ./etc/ktulu/tests/hello.json
fizz 0.7.1-X (20200814.0954) [lnx.x64|8|1]
Press the ESC key at anytime for input prompt

load : loading ./etc/ktulu/tests/hello.json ...
load : loaded ./mod/lnx/x64/modCTM.so in 0.000s
load : loading ./etc/ktulu/tests/hello.fizz ...
load : loaded ./etc/ktulu/tests/hello.fizz in 0.003s
load : loading completed in 0.005s
1 []
2 []
3 []
4 []
5 [[ctm.proc.sheldon, 4, enkoogy, tweet, bazinga, 1, 1, 1]]
6 []
7 [[ctm.proc.sheldon, 6, vfobcwu, tweet, bazinga, 0.500000, 0.500000, 1]]
8 []
9 [[ctm.proc.sheldon, 9, kszygwio, tweet, bazinga, 0.875000, 0.875000, 1]]
10 []
11 [[ctm.proc.sheldon, 11, kyygvic, tweet, bazinga, 1, 1, 1]]
12 []
13 [[ctm.proc.sheldon, 13, jvhyrins, tweet, bazinga, 0.625000, 0.625000, 1]]
14 []
15 [[ctm.proc.sheldon, 15, hqvkskji, tweet, bazinga, 0.875000, 0.875000, 1]]
16 []
```



Note how the weight of the *chunk* change as the *processor* take into account the *DROP operator*. Let's now add a second *processor* in `hello.fizz`:

```
1 ctm.proc.leonard {
2   class = CTMCBFSActor,
3   dreco = 4,
4   weight = 1,
5   mood = 1
6 } {
7
8   () :- @ctm.proc.tick(_,_),
9         rnd.real(1,_?[gt(0.2)]),
10        ~self(tweet,{},[oh_boy,$weight,$mood]);
11
12 }
```

It is setup pretty much the same way, except for the *gist* and for the likelihood of it posting a *tweet*. Let see how that plays out:

```
jlvr@korok:~/Code/okb/apps/ktulu$ ./fizz.x64 ./etc/ktulu/tests/hello.json
fizz 0.7.1-X (20200814.0954) [lnx.x64|8|1]
Press the ESC key at anytime for input prompt

load : loading ./etc/ktulu/tests/hello.json ...
load : loaded ./mod/lnx/x64/modCTM.so in 0.000s
load : loading ./etc/ktulu/tests/hello.fizz ...
load : loaded ./etc/ktulu/tests/hello.fizz in 0.005s
load : loading completed in 0.006s
1 []
2 []
3 [[ctm.proc.leonard, 3, bhddympv, tweet, oh_boy, 1, 1, 1]]
4 []
5 [[ctm.proc.sheldon, 5, xxxguhyn, tweet, bazinga, 1, 1, 1]]
6 []
7 [[ctm.proc.leonard, 7, rcatiahh, tweet, oh_boy, 0.875000, 0.875000, 1]]
8 []
9 [[ctm.proc.leonard, 9, ctgcsqws, tweet, oh_boy, 1, 0.937500, 1]]
10 []
11 [[ctm.proc.leonard, 11, ehvycet, tweet, oh_boy, 0.625000, 0.625000, 1]]
12 []
13 [[ctm.proc.sheldon, 12, featlusi, tweet, bazinga, 1, 0.875000, 1]]
14 []
15 [[ctm.proc.leonard, 14, bsmqfrkc, tweet, oh_boy, 1, 1, 1]]
16 []
17 [[ctm.proc.sheldon, 17, mvfhfpib, tweet, bazinga, 0.875000, 0.750000, 1]]
18 []
19 [[ctm.proc.sheldon, 19, nmsusiee, tweet, bazinga, 1, 0.937500, 1]]
20 []
21 [[ctm.proc.leonard, 21, avaqkgat, tweet, oh_boy, 1, 0.812500, 1]]
22 []
23 [[ctm.proc.leonard, 23, dsnkrrnp, tweet, oh_boy, 0.625000, 0.625000, 1]]
24 []
25 [[ctm.proc.leonard, 25, cllllvqk, tweet, oh_boy, 0.875000, 0.875000, 1]]
26 []
27 [[ctm.proc.leonard, 27, qkvqonjj, tweet, oh_boy, 1, 1, 1]]
28 []
29 [[ctm.proc.sheldon, 29, ejmdtabr, tweet, bazinga, 1, 0.812500, 1]]
```

## Internal Language Representation (ILR)

By convention, processors communicate primarily between each others by exchanging tweets and queries represented in *functor* based expressions. Answers to *queries* can be simple *terms* such as *number*, *string*, *list*, *symbol*, *frame* or *functor*.

As *functors* can be nested, a *processor* must have the ability to resolve *functor* terms, or it should only pay attention to *functors* who's depth is 1 (meaning there's no nested *functor*) and relies on another *processor* to resolve them (see `ilr.fizz` and the `nouns` test).

A *frame* is to be used to represent more detailed entity. The `token` label will hold the value as an *atom* that best represent the entity (for example *fox*). The label `lexic` can be used to indicate the lexical category of the entity if the entity isn't already known. The label `slots` will be used to hold a collection (a *frame* of properties known (or queried) for the entity. For example:

```
{
  token = fox,
  lexic = noun^c,
  slots = {
    speed = quick,
    color = brown
  }
}
```

This will also be valid, for example as the result of NLP:

```
{
  token = fox,
  lexic = noun^c,
  slots = {
    adj = [quick, brown]
  }
}
```

The supported main lexical meaning of an entity are:

<b>pronoun</b>	substitutes for a noun or noun phrase
<b>noun^a</b>	an abstract noun for ideas, concepts, feelings, and traits.
<b>noun^c</b>	a common noun for person, animal, place, thing, or idea
<b>noun^p</b>	a proper noun for identifies someone or something, a person or a place
<b>concrete</b>	represents a thing that is real and tangible
<b>abstract</b>	represents a thing that is more like a concept or idea
<b>adjective</b>	represents something that describes its referent
<b>verb</b>	represents a verb to be treated as a noun

The presence of the label `fuzzy` can indicate, via a numerical value (between 0 and 1) the truthness/confidence in the entity as described. Lastly, the label `slots` can be used to provide a *frame* which contains adjectives. The label `det` can be used to indicate the determinant of the entity (e.g. *the*, *this*). In the case of a sub-expression, the label `type` will be used to indicate the type of the sub-expression (e.g declarative, imperative ...).

ILR expressions are expected to conform to the following format:

`label(frame,frame|list)`

Where *label* describes a verb or a form, the first *frame* is a set of attributes (for example the tense). The second *term* is either a *frame* or a *list* (in the case of a form). It is a collection of (labeled, in the case of a verb) arguments. How a specific *functor* is handled by a *processor* is up to the *processor*. However, consistency within a single system should be considered.

While an given expression's attributes can be anything, the following set of attributes are to be supported:

<b>mod</b>	Provides a modifier, usually for a verb (such as: <i>for</i> in <i>look for</i> )
<b>tense</b>	Indicates the tense at which the relation hold using a frame with: <b>tense:</b> <code>present, future, past</code> <b>tweak:</b> <code>indicative, completed, continuous, continued, conditional</code> <b>stamp:</b> an actual (or a range) timestamp
<b>fuzzy</b>	The fuzzy value of the expression (0 = false, 1 = true)
<b>query</b>	When the expression is used in a <i>query</i> , a symbol may indicate what type of the query it is (e.g. <i>who</i> ).
<b>genre</b>	top most <b>functor</b> can have this attributes to indicates the genre of the complete expression, such as: <b>dec</b> for declarative expression. <b>int</b> for interrogative expression. <b>imp</b> for imperative expression. <b>exc</b> for exclamative expression.

Both *verb* and *form* can be anything meaningful to the system. They do need to be known to the system. Some of the *form* are already predefined. For instance:

<b>not</b>	the negation of an expression.
<b>and</b>	a conjunction expression for cumulative non-contrasting <i>terms</i> .
<b>or</b>	a conjunction expression for alternative non-contrasting <i>terms</i> .
<b>junc</b>	a silent conjunction expression for cumulative non-contrasting <i>terms</i> .
<b>but</b>	a conjunction expression for adversative, presents a contrast or exception
<b>owned</b>	an <i>term</i> (first) is owned by another <i>term</i> (second).
<b>would</b>	indicates the consequence of an imagined event or situation ( <i>single term</i> ).
<b>count</b>	indicates a known (first <i>term</i> ) quantity of something (second <i>term</i> ).
<b>to</b>	indicates that the first <i>term</i> is needed to achieve then second <i>term</i> .

## Natural Language Processing (NLP)

Within the context of this project, NLP is defined as the decoding of a *natural language* (English) sentence and re-encoding of it as an ILR expression. It is provided by a set of *fizz* code contained in the `./etc/ktulu/nlr/nlp` folder and relies and some augmentable lexicon to perform the identification of the tokens that compose a sentence.

The main *elemental* objects to be used are `nlp.decode` and `nlp.decode.all`. The first one will provides all possible parsing solutions in the typical *fizz* way with the second will combine all the solutions in a *list*.

Before they can be used, there's two *elementals* that must be defined as part of the *fizz* solution you will be using: `nlp.noun.stem.extd` and `nlp.verb.stem.extd`. They both provides a way to specify custom stemming for nouns and verbs. Just defines them as follow for now:

```
nlp.noun.stem.extd { // noun stemming (extended)
    no.match = fail
} {
```

```

}

nlp.verb.stem.extd { // verb stemming (extended)
  no.match = fail
} {

}

```

Let's now see a couple of examples:

```

?- #nlp.decode("You are a robot.",:f)
-> ( be({tense = {tense = present, tweak = indicative}, genre = dec}, {subject = {lexic = pronoun~n
, token = you}, object = {lexic = [noun~c, noun~p], token = robot, maybe = yes, slots = {det =
{lexic = article, token = a}}}}) ) := 1.00 (0.087) 1
?- #nlp.decode("You are a nice robot.",:f)
-> ( be({tense = {tense = present, tweak = indicative}, genre = dec}, {subject = {lexic = pronoun~n
, token = you}, object = {lexic = [noun~c, noun~p], token = robot, maybe = yes, slots = {adj =
{token = nice, maybe = yes, lexic = adjective}, det = {lexic = article, token = a}}}}) ) :=
1.00 (0.120) 1

```

For each possible parsing of the string, the variable `f` will be bound to a *functor* which is the corresponding ILR expression. For this simple sentence, we then have the `be` verb with two slots: a `subject` and an `object`. Note, that the token `robot` isn't part of the lexicon and thus marked with a `maybe` attribute and have for `lexic` a *list* of possibilities. Adjectives to a noun are assembled (into a *list* if there's more than one)

Multiple parsing solutions is often the result of ambiguities in the natural language expression. Here's an example: *Do I need to throw the trash in the bin or throw the trash that is already in the bin?*

```

?- #nlp.decode("Throw the trash in the bin.",:f)
-> ( throw({tense = {tense = present, tweak = indicative}, genre = imp}, {object = {lexic = noun~c,
token = trash, slots = {det = {lexic = determiner, token = the}}}, in = {lexic = noun~c, token
= bin, slots = {det = {lexic = determiner, token = the}}}}) ) := 1.00 (0.359) 1
-> ( throw({tense = {tense = present, tweak = indicative}, genre = imp}, {object = {lexic = noun~c,
token = trash, slots = {in = {lexic = noun~c, token = bin, slots = {det = {lexic = determiner,
token = the}}}, det = {lexic = determiner, token = the}}}}) ) := 1.00 (0.367) 2

```

We can here use `nlp.decode.all` to assemble a *list* of all solutions:

```

?- #nlp.decode.all("Throw the trash in the bin.",:f)
-> ( [throw({tense = {tense = present, tweak = indicative}, genre = imp}, {object = {lexic = noun~c
, token = trash, slots = {det = {lexic = determiner, token = the}}}, in = {lexic = noun~c,
token = bin, slots = {det = {lexic = determiner, token = the}}}}), throw({tense = {tense =
present, tweak = indicative}, genre = imp}, {object = {lexic = noun~c, token = trash, slots = {
det = {lexic = determiner, token = the}}}, in = {lexic = noun~c, token = bin, slots = {det = {
lexic = determiner, token = the}}}}), throw({tense = {tense = present, tweak = indicative},
genre = imp}, {object = {lexic = noun~c, token = trash, slots = {in = {lexic = noun~c, token =
bin, slots = {det = {lexic = determiner, token = the}}}, det = {lexic = determiner, token = the
}}}}), throw({tense = {tense = present, tweak = indicative}, genre = imp}, {object = {lexic =
noun~c, token = trash, slots = {in = {lexic = noun~c, token = bin, slots = {det = {lexic =
determiner, token = the}}}, det = {lexic = determiner, token = the}}}})] ) := 1.00 (0.502) 1

```

When parsing fails to provide any solutions, it is often due to an incomplete lexicon. As we have seen, the parser can handle some unknown tokens but not verbs. The best approach when dealing with potentially unknown tokens, is to request from `nlp.decode` to provide a list of all unknown tokens. You can then decide

if the end user should be requested to fill in the lexicon gap.

If the first *term* used with a `nlp.decode predicate` is a *frame*, it can be used to request it to output not the parsing, but the result of the tokenization and identification:

```
?- #nlp.decode({out=toks},"Fly me to the moon!",:f)
-> ( [{lexic = unknown, token = fly}, {lexic = pronoun~n, token = me}, {lexic = [adverb,
    preposition], token = to}, {lexic = determiner, token = the}, {lexic = unknown, token = moon},
    {lexic = punctuation, token = exclamation}] ) := 1.00 (0.069) 1
```

We can also, just get the unknown tokens without attempting to parse the sentence, like this:

```
?- #nlp.decode({out=toks},"Fly me to the moon!",_,:u)
-> ( [{lexic = unknown, token = fly}, {lexic = unknown, token = moon}] ) := 1.00 (0.067) 1
```

As `nlp.decode.all` has a built-in *time-to-live*, we can retrieve the list of unknown token when the parsing failed:

```
?- #nlp.decode.all("Fly me to the moon!",:f,:u)
-> ( [] , [{lexic = unknown, token = fly}, {lexic = unknown, token = moon}] ) := 1.00 (1.615) 1
```

The *time-to-live* duration can be adjusted, by using as first *term* a frame as such:

```
?- #nlp.decode.all({ttl=0.5},"Fly me to the moon!",:f,:u)
-> ( [] , [{lexic = unknown, token = fly}, {lexic = unknown, token = moon}] ) := 1.00 (0.568) 1
```

To extend the lexicon used by the parser, the following *elementals* can have their definition updated or augmented by adding separate but identically named *elementals*:

```
nlp.adjective
nlp.adverb
nlp.article
nlp.compound
nlp.conjunction
nlp.idiom
nlp.intensifier
nlp.abstract_noun
nlp.common_noun
nlp.proper_noun
nlp.number
nlp.preposition
nlp.pronoun
nlp.verb
```

Except for `nlp.verbs` they all require the same number of *terms*: (*frame*,*symbol*,*frame*)

Where the first *frame* will be provided by the parser with some context on the on-going parsing and can be used to help identify the second *term* which is the tokenized *symbol* from the *string*. The third *term* is the *output* and is expected to be a *frame* containing at the very least the lexical type of the token and its associated *symbol*. For example:

```
(_,bob,{lexic=noun~p,token=bob});
(_,seventeen,{lexic=number~c,token=17});
(_,smooth,{lexic=adjective,token=smooth});
```

For `nlp.verb`, four *terms* are expected: (*frame*, *symbol*, *list*, *list*)

The first *term* serves the same purpose as for the rest of the lexicon *elementals*. The second *term* is a *symbol* that identifies the verb. The third is a *list* which indicates if the verb is transitive (**t**), intransitive (**i**), distransitive (**d**) and/or passive (**p**). The fourth *term* is a *list* containing the known form of the verb: present, preterite, past principle. For example, the verb **chose** will be defined as such:

```
(_, chose, [t,i], [chose, chose, chosen]);
```

## Natural Language Generation (NLG)

Generation of *natural language* (English) is handled by a set of *fizz* code contained in the `./etc/ktulu/nlr/nlg` folder. It is a non-conscious process which takes ILR and generates sentence(s) (strings) from it using augmentable knowledge.

The functionality is provided by the *elemental*: `nlg.encode` which converts an ILR expression into a *string*. To process an ILR expression, a *predicate* must be used with two input *terms* providing the type of the expression and the expression it-self. The third *term* will be unified with the generated *list*.

The type of expression supported at the moment are:

<code>dec</code>	for declarative expression.
<code>int</code>	for interrogative expression.
<code>imp</code>	for imperative expression.
<code>exc</code>	for exclamative expression.

For example:

```
?- #nlg.encode(like({tense = {tense = present, tweak = indicative}, genre = dec}, {subject = {lexic  
    = pronoun^n, token = i}, object = {lexic = noun^c, token = ice_cream}}),:s)  
-> ( "I like ice cream." ) := 1.00 (0.133) 1
```

At this time, NLP and NLG use different source of lexical knowledge. This will be merged in the near future. Meanwhile, you may need to extend the following *elementals*:

<code>nlg.verbs.extd</code>	For extra verbs to get correctly conjugated, they must be known using this label, where each new verb must be defined with 4 terms: its symbol, the actual verb, simple past form and past participle form for example: ( <b>grab</b> , <b>to_grab</b> , <b>grabed</b> , <b>grabed</b> )
<code>nlg.nouns.extd</code>	To avoid having to provide the lexical attribute of a <i>symbol</i> , this label can be augmented to define it for a symbol. For example: ( <b>Bob</b> , <b>proper</b> )
<code>nlg.symbol.plural.excep.extd</code>	For nouns that have a plural exception, this label should be used to define it. It simply match a given <i>symbol</i> with its plural form, e.g. ( <b>tooth</b> , <b>teeth</b> )