fizz

Jean-Louis Villecroze
`jlv@f1zz.org` `@CocoaGeek`

September 12, 2023 (*Version 0.8.6-X*)

Abstract

fizz is an experimental language and runtime environment for the exploration of *cognitive architectures* and combined *Machine Learning* (ML) and *Machine Reasoning* (MR) solutions. It is based primarily on *symbolic programming* and *fuzzy formal logic*, and it features a distributed, concurrent, asynchronous and responsive *inference engine*.

Contents

1 About this document

This document is a user manual for *fizz* and assumes some basic familiarity with *logic programming*. It is divided into the following parts:

Concepts & Syntax	introduces the concepts and the syntax used to describe and manipulate *knowledge*
Console	introduces the usage of the builtin *console*
Terms	introduces the various types that can be manipulated
Primitives	lists and describes all the *primitives* functions
Elementals	lists and describes all the supported *Classes* of *Elementals*
Advanced topics	describes more advanced topics including the *Services*
Release notes	contains pertinent information for each subsequent releases

All code elements are presented in a distinct font like `print("hello, world!")`. Note that any tabulation shown in a listing is only present to enhance the readability of the code. Tabulations are not part of the language syntax. *Primitives* syntax is often a combination of code element and italic font. The part in italic is always the input to the primitive. *Primitives* inputs use special symbols :

symbol?	indicates that the input is optional
symbol|*number*	indicates that the input can be either a *symbol* or a *number*
symbol+	indicates that the *primitive* can take on several *symbols* as input, but at least one is required
*symbol**	indicates that the *primitive* can take on several *symbols* as input, but one is optional

Many thanks to Joshua Nozzi (@JoshuaNozzi) and Keith Kolmos (@KeithMKolmos) for reviewing this document and providing many insightful corrections and suggestions.

2 Concepts & Syntax

If you are familiar with *PROLOG*, you will find that *fizz* takes some of its fundamental elements and syntax from it. There are five main concepts in *fizz* , which we will be discussing in this section:

Knowledge	is a collection of related *statements* and/or *prototypes*.
Statement	is a collection of *terms* with an assigned *truth value* (think *fact*).
Predicate	is a labeled collection of *terms* with an assigned *truth value* range (or *variable*).
Prototype	is a chained collection of *predicates* that can be evaluated (think *rule*).
Elemental	is a runtime object which hold *knowledge* and can answer to query.
Service	is a runtime object which provide a unique service within the *runtime*.

One of the main differences between *PROLOG* and *fizz* is how *inference* is done not by a single entity having access to all *facts* and *rules*, but by the cooperation of a collection of objects each having access only to what they must know (*knowledges*). *Elemental* objects in *fizz* are very much independent *actors*, which must exchange messages (mostly by a queries and replies mechanism) in order to execute any inferences. While this is far from being the most efficient method (and performance in some aspect is much worse for some types of inferences) it allows for instance a *statement* that is broadcasted to trigger the execution of any *prototype* that references it (via a *predicate*). It also supports inferences to be distributed among many cores and/or many hosts.

2.1 Knowledge

A *Knowledge* groups a series of related *Statements* and *Prototypes* under the same logical concept (often refered in this document as "label"). For example, if we wanted to create a list of the three basic colors we would define it as follows:

```
color {

    (red,1.0,0.0,0.0);
    (green,0.0,1.0,0.0);
    (blue,0.0,0.0,1.0);

}
```

Knowledge definition always starts with a label that identifies the concept, followed by a *frame* (optional) and a series of *Statements* and/or *Prototypes* within curly brackets. The *frame* (see Section 3.4 on page 13 for details on that *term*) specified after the *symbol* is known as the *properties* of the *knowledge*.

When a *knowledge* is used to define only *statements*, it is said to be *factual knowledge*. If it contains only *prototypes*, it is called a *procedural knowledge*.

2.2 Statement

A *Statement*, as we have seen in the example above, is a comma-separated list of *terms* within parantheses and terminated by a semicolon. We will look into all the supported *terms* in more details in Section 3 on page 9, but so far we have used *symbols* and *numbers*. Each time a *statement* is defined, it can be assigned a *truth value* (indicating the relation of the *statement* to truth). Let's look at an example where each *statement* is assigned a value to represent the likelihood of a given weather occurence in a particular city:

```
weather {

    (paris,rain)        := 0.8;
    (seattle,sunny)     := 0.2;
    (london,fog)        := 0.9;
    (mawsynram,rain)    := 1;
    (honolulu,snow)     := 0;
    (honolulu,rain)     := 0.1;
    (honolulu,sunny)    := 0.6;
    (honolulu,cloudy)   := 0.3;

}
```

It's so unlikely that you will see snow in Honolulu, that we here state that such statement is false.

A *truth value* is always a number between 0 (false) and 1 (true). When no *truth value* is assigned, the default value for a *statement* is 1. It is always defined last, prefixed with a `:=`. As part of a *statement* definition, we could also join a collection of *properties* that apply to the *statement* in the form of a *frame* object which is inserted right after the closing parenthesis. Here's a version of the above knowledge where each statement have been timestamped (see section 5.2 on page 43 for how):

```
weather {

    (paris,rain)        {stamp = 1507093154.766867} := 0.8;
    (seattle,sunny)     {stamp = 1507093158.846844} := 0.2;
    (london,fog)        {stamp = 1507093174.863446} := 0.9;
    (mawsynram,rain)    {stamp = 1507093176.743262} := 1  ;
    (honolulu,snow)     {stamp = 1507093177.671228} := 0  ;
    (honolulu,rain)     {stamp = 1507093178.743266} := 0.1;
    (honolulu,sunny)    {stamp = 1507093179.807307} := 0.6;
```

```
    (honolulu,cloudy)   {stamp = 1507093180.879415} := 0.3;

}
```

Without getting ahead of ourselves (next section), a *statement*'s properties can be queried the same way as its *terms*:

```
?- #weather(:x,:y) {stamp = :s?[gte(1507093176)]}
-> ( mawsynram , rain , 1507093176.743262 ) := 1.00 (0.001) 1
-> ( honolulu , rain , 1507093178.743266 ) := 0.10 (0.002) 2
-> ( honolulu , sunny , 1507093179.807307 ) := 0.60 (0.002) 3
-> ( honolulu , cloudy , 1507093180.879415 ) := 0.30 (0.002) 4
```

2.3 Predicate

A *Predicate*, while being syntactically similar to a *Statement*, represents not a *fact* but a *question* to be figured out. In the following example we will write a *predicate* which formulates the query: *"tell me where it is very likely to rain"*:

```
@weather(:x,rain) = <0.7|1.0>
```

The `<0.7|1.0>` at the end of the *predicate* is a *truth value range*. In this case, it indicates that we will only accept the *statements* where *truth values* are between `0.7` and `1.0`. Beside a *range*, a *predicate* will also accept a *number* or an unbound *variable*. The latter will allow the *truth value* of each *statements* received for the *predicate* to be used in the following *predicates*.

Because a `predicate` is querying a particular *knowledge*, its *label* must be indicated. Here, we're using the `weather` *knowledge* we defined earlier. The `@` prefix indicates to the runtime that the predicate is referencing a *knowledge* and not a *primitive*. *Primitives* are built-in functions, such as `lst.length`, which can be used to get the number of elements in a list *term*. See Section 5 on page 40 for all the supported *primitives*. If we wanted to use a *primitive* we would have omitted the `@` like in this example:

```
lst.length([1,2,3,4,5],:length)
```

There is however a situation when a prefix (other than `!`) can be used with a *primitive*. Using `&` will cause the *primitive* to be executed on the *runtime environement* threads pool and not within the *elemental*. We will often reference this as "offloading".

A secondary meaning of the `@` prefix is to indicate that the *predicate* should be considered a *trigger*. As stated in section 2 on page 2), when a *statement* is broadcasted in the *runtime environment*, the *predicate* will set up the *prototype* to which it belongs for evaluation. For performance reasons, it is often best to indicate when a given *predicate* is not a *trigger*. For these situations, the `@` prefix can be replaced by `#`. If we look back at our earlier example, any new `weather` *statement* will activate the *prototype* in which we used that *predicate*, we can change it as follow:

```
#weather(:x,rain) <0.7|1.0>
```

`~` is another prefix that can be used for a *predicate*. When used in conjunction with the *predicate* label `self`, it indicates a self referencing *predicate* (a recursive *predicate*). When using with any other *elemental* label, it will cause the *query* to be sent to one (picked randomly) of the *elemental* in the *substrate* with that label.

Using `self` instead has the advantage of being often shorter to type and to enable the *elemental* to be cloned since such *predicate* will always point to the right *elemental*. For example, here's an *elemental* which calculate the sum of a all the *numbers* in a *list*:

```
lst.sum {

    ([],0)^         :- true;
    ([:h],:h)^      :- true;
    ([:h|:r],:s)    :- ~self(:r,:s.r), add(:h,:s.r,:s);

}
```

The difference between `#self` and `~self`, is that when the tilde is use, the *predicate* will only be send to the *elemental* itself. No other *elemental* with the same label will get the query.

The fourth prefix that can be used with *predicates* is `*`. When used, the *query* will *round-robin* between all *elementals* that can answer the *query*. This prefix allows *queries* to be distributed amongst multiple *elementals*, potentially executing concurrently on different CPUs.

The fifth and final supported prefix is `?`. when used, the *query* will continue even in the case where the *predicate* fails. When the truth value of the *predicate* is also inspected (by assiging it to an unbound *variable*), using this prefix allows for a custom handling of a failure as seen in this example:

```
maybe.number {

    (:x,:v) :- ?is.number(:x) = :v;

}
```

Lastly, if a caret (`^`) is added right after the *terms* of the *predicate*, it will indicate that once the *predicate* as succeded, the solver should not consider any other alternative based on any of the *predicates* that came before (this is similar to the `cut` operator in *PROLOG*). When the *predicate* is part of series of *prototypes*, the other *prototypes* may still be considered depending on what type of *predicates* came before the *cut*. To illustrate a *cut* let's consider the following example which defines `str.default` as a *knowledge* which given a *term* will either "return" that *term* when it is a valid *string* or a second *term* if it is not:

```
str.default {

    (:a,:b,:b) :- console.puts("1>"), !is.string(:a)^;
    (:a,:b,:b) :- console.puts("2>"), is.string(:a) , str.length(:a,0)^;
    (:a,:b,:a) :- console.puts("3>"), is.string(:a) , str.length(:a,_?[gt(0)]);

}
```

If we now query this *knowledge* with a *symbol* as first *term*, we would expect the second *term* to be unified with the third *term*:

```
?- #str.default(a,"b",:b)
1>
-> ( "b" ) := 1.00 (0.001) 1
```

As we have started each *prototypes* with a call to the `console.puts` *primitive*, we can observe how the second and third *prototypes* were indeed not called. Have we had omitted the *cut* from the two first *prototypes*, we would have seen this:

```
?- #str.default(a,"b",:b)
1>
2>
-> ( "b" ) := 1.00 (0.001) 1
3>
```

Because each of the *prototypes* is composed of *primitives* only, they will be considered sequentially by the solver. In fact, the solver will always considere *prototypes* sequentially but if a *predicate* is not a *primitive*, the following *prototype* will be considered while the solver waits for answers to the *query* it put out for the *predicate*.

As we would expect, if the *cutting predicate* is not reached by the solver the *cut* will have no effect as we see in the following example:

```
?- #str.default("a","b",:b)
1>
2>
3>
-> ( "a" ) := 1.00 (0.001) 1
```

As you probably noticed in the past examples, we have used as one of the *terms* `:x` and `:length`. These are *variables* and they can stand for any other type of *terms* (except *variables* themselves) during the *inference* process. See Section 3.8 on page 16 for more details on *variables*.

2.4 Prototype

A *Prototype* defines the relationship between a collection of *statements*, which may produce a new *statement* if the logical inference reaches a conclusion. For example, we could create a new logical concept that would contain a *prototype* based on the `weather` example we wrote earlier. We will call it `surely_raining`:

```
surely_raining {

    (:x) :- @weather(:x,rain) = <0.7|1.0>;

}
```

A *prototype* is composed of an *entrypoint*: a comma-separated list of *terms* within parentheses followed by a `:-` and a comma-separated collection of *predicates* terminated by a semi-colon. The *entrypoint* specifies what a *predicate* referencing this *knowledge* would be like and it is also used during *inference* to check if the *prototype* should be used. In this case, it would have a single *term* that will be unified with the local variable `:x`. If we wanted to check if it is surely raining in Paris, we would write:

```
@surely_raining(paris)
```

If a caret (`^`) is insterted between the *entrypoint* and the `:-`, it will indicates that during inferences when the *prototype*'s *entrypoint* unifies with a *statement* or a query, no other *prototypes* should be considered, even if, in the end, the inference fails. This allows for cases where a single *prototype* among many must be used.

In some instances, it's often desired to take the negation of a *predicate*. This can be done by prefixing the *predicate* with a `!` like this:

```
!is.string(3.14)
```

Since `3.14` is a number, the call to the *primitive* `is.string` will return a *truth value* of `0` since that *primitive* checks if its argument is a *string*. Negating this will result in the *predicate* returning `1` as its *truth value*. When a *prototype* contains more than a single *predicate*, the *truth value* of the statements matching each *predicate* will be used to compute the *truth value* of the *predicate* as a *fuzzy logical and*. For example, to answer the question *"Where are we the most likely to see a rainbow?"* we would write a new *knowledge* as follows:

```
maybe_rainbow {

    (:x) :- @weather(:x,rain), @weather(:x,sunny);

}
```

With the `weather` *knowledge* we have, we would get the answer `honolulu` with a *truth value* of `0.1`.

Before moving on to the next concept, lets backtrack to the following example:

```
surely_raining {

    (:x) :- @weather(:x,rain) = <0.7|1.0>;

}
```

The *prototype* could have been written using a constrained *wildcard variable*:

```
surely_raining {

    (:x) :- @weather(:x,rain) = _?[lte(1.0),gt(0.7)];

}
```

Using a *variable* would have allow us to take in the actual *truth value* of all the *statements* satisfying the *predicate* and use them in whichever way necessary.

Prototypes using `:-` evaluates their *truth value* by performing a *fuzzy and* which takes the minimum value of all *predicates*. This behavior can be changed to a *fuzzy and* where the *truth value* of each *predicates* are multiplied to each other by using `&-` instead. *fuzzy or* can be selected using `|-`, it will compute the *truth value* by adding all the *truth values*. Unlike with the commmon `:-` which will stop evaluating its *predicates* once one evaluate to *false*, the two evaluation modes just described will evaluate every *predicates*.

Here's an example where an *animal* is either a *dog*, a *cat* or a *duck*:

```
animal {
    (:x)    |-  #dog(:x), #cat(:x), #duck(:x);
}

dog {
    no.match = fail
} {
    (fido);
    (spot);
    (rover);
}

cat {
    no.match = fail
} {
    (kitty);
    (kelly);
}

duck {
    no.match = fail
} {
    (donald);
    (daffy);
    (huey);
}
```

As expected, querying *animal* will give us:

```
?- #animal(daffy)
-> (  ) := 1.00 (0.001) 1
```

Note that a similar behavior could be have obtained by using the *cascade* mode as follow:

```
animal2 {
    cascade = yes
} {

    (:x)    :-  #dog(:x)^;
    (:x)    :-  #cat(:x)^;
    (:x)    :-  #duck(:x)^;
    (:x)^   :-  false;

}
```

2.5 Elemental

Elementals in *fizz* are the main components of the *runtime environment* (also called *substrate*). In most cases, when a *knowledge* is loaded a new *elemental* object is created to handle it, however a single *elemental* can manage multiple *knowledges*. There are several types of *elementals* in *fizz* . See Section 6 on page 110 for more details. Each *elemental* presents on the *substrate* is assigned an unique identifier (GUID), unless one is provided.

Elementals objects can have *properties* associated with them. In most cases, such data allow for customization or optimization of the objects. This is done with a *frame* (which is a supported *term*, see Section 3.4 on page 13) in between the *knowledge*'s body and its label, as seen in the following example:

```
rand {class = MRKCRandomizer, min = 1550, max = 1650} {

}
```

In the example we request a specific class of *elemental* object to be instantiated using the `class` label and specify a `min` and `max` value. While these two *properties* are specific to `MRKCRandomizer`, `class` is a reserved label. There's a few other reserved labels:

`alias`	a *symbol* by which the *elemental* will also be known locally
`class`	a *symbol* indicating the *class* of the *elemental* object
`clone`	a *symbol* indicating the *elemental* object to be used as the model
`guid`	a *string* containing the GUID to be used by the *elemental* object
`spawn`	assigned to the *symbol* `no` will not cause the *knowledge* to instantiate a new *elemental*
`nosy`	when set to `yes` (the default), any reply to a query that wasn't initiated by the *elemental* will be checked to see if it can be used as trigger.
`chatty`	when set to `yes` (the default), the *elemental* will publish the *statements* it uses as replies to queries.
`ttl`	when set, the *elemental* will use the value (in seconds) as the value for the TTL of any queries it send out instead of the global value.

An *elemental*'s properties can be accessed at runtime by any *prototype* being executed by the *elemental*. Either by using the `primitives peek` and `poke` (see Section 5.2 on page 51) or by using the *constant* access syntax (e.g. `$guid`). When using the *constant* form, the label of the *elemental* can be retreived at runtime with `$self`.

If there is no existing matching *elemental* for a *knowledge* (that is, no *elemental* objects with the same name and capable of accepting the *knowledge*), a new one will be instantiated even if `spawn` is set to `no`. If the `clone` property is given, the first *elemental* that answers to that label will be cloned and any properties specifies in the source *elemental* will be replaced by the value in the target *elemental*.

Depending on the situation, setting the properties `nosy` and `chatty` to `no` can help improve the performances of the system by lowering the unecessary background inferring.

2.6 Service

Services are a special case of *elemental* objects which exist on the *substrate* as a singleton. Each of these objects provides *services* to all other *elementals*. The *services* are provided via the classic query/reply pattern shared by all *elementals*. See Section 8 on page 148 for more details.

3 Terms

There are 11 categories of *terms* in *fizz* . In this section we will introduce each one of them and see how they are each different from the other. They all have one thing in common, however: their immutability. While this may be common with *atoms*, it is less common with more complex data such as *lists* (at least in non-functional languages).

3.1 Atoms

There are different kinds of *atoms* in *fizz* :

- Number
- String
- Symbol
- Binary
- Guid

They are the most basic data that can be handled.

3.1.1 Number

A *number* in *fizz* represents a 64-bit numerical value. It can be an integer (signed or unsigned) or a floating point value, depending on how it is written and eventually postfixed. For example, if we consider the following statement:

```
yearly_stats {

    (2001,0.4,45u,3f);

}
```

The first *term* will be understood as a signed integer, the second term will be floating point, while the third *term* will be unsigned. The last *term*, by the addition of the postfix `f`, will be promoted from signed integer to floating point. Numbers expressed in *scientific notation*, such as `3e-2` will also be understood as floating point values. For two numbers to be successfully unified, their difference must be smaller than the *epsilon* value specified in the *runtime environment* configuration (see Section 4.2 on page 23).

3.1.2 String

Strings in *fizz* are no different from other languages: a series of characters between double quotes. For example:

```
quotes {

    (DrSeuss,"Don't cry because it's over, smile because it happened.");
    (OscarWilde,"Be yourself; everyone else is already taken.");
    (Gandhi,"Be the change that you wish to see in the world.");

}
```

The common escape sequence using a backslash (for example `"\n"`) is supported with the following characters:

a	alert (bell) character
b	backspace
f	formfeed
n	newline
r	carriage return
t	horizontal tab
v	vertical tab

Two strings will only unify if their content and length perfectly match. Note that at this time, Unicode isn't supported.

3.1.3 Symbol

Symbols in *fizz* are fundamental. Just like *strings*, they can contain characters as well as numbers but they are not started and terminated by double quotes. As such, they cannot contain spaces, nor start with a number. They are often used as identifiers. Here are a few example of valid *symbols*:

```
identifiers {

    (jill);
    (jack74);
    (bob.phone);
    (bob.age);

}
```

Two *symbols* will only unify if they perfectly match.

3.1.4 Binary

Binary terms are a way for *fizz* to handle *elementals* specific binary data. Such *terms* uses `base64` to encode binary contents into a string, and they are specified in *fizz* code using a single quoted functor as in the following example:

```
blobs {

    ('binary("dGhlIGJyb3duIGZveCBqdW1wcyBvdmVyIHRoZSBsYXp5IGRvZw=="));

}
```

Two *binaries* will only unify if there's a perfect match of the decoded binary data. When a *knowledge* containing such *term* is parsed, the parsing will fail if the binary data fails to be decoded.

3.1.5 Guid

Guid terms are a way to represent globally unique identifier. Such *terms* are specified in *fizz* code using a single quoted functor as in the following example:

```
guids {

    ('guid("71cfade6-3cab-c34e-3ca6-e7a43e6fb5f7"));

}
```

3.2 List

Lists are common and widely used. They allow the grouping a collection of *terms* into a single *object*. The syntax for a *list* is a comma-separated collection of *terms* (including *lists*) in between square brackets. For example, we could have written the *color* example from earlier where each colors RGB values are expressed as *lists*:

```
color {

    (red,[1.0,0.0,0.0]);
    (green,[0.0,1.0,0.0]);
    (blue,[0.0,0.0,1.0]);

}
```

There's a special kind of *list* that can be used to *split* the content of the *list* (head and rest). Used with recursion, it makes it possible to iterate over all the *terms* in a *list* possible. Consider the following *knowledge*:

```
lst.print {

    ([]);
    ([:h|:r]) :- console.puts(:h), @lst.print(:r);

}
```

The above example sets up `lst.print` with a *prototype*, which will print the *head* of the *list* and then recursively call itself with the *rest* of the *list*. The *knowledge* also constains a *statement* for when the *list* is empty. While it is not mandatory, it will cause a call to `lst.print` to always succeed.

3.3 Data

Data terms are less practical but more efficient than a *list* to use when a large number of values of the same type must be held and processed. Such *terms* are specified in *fizz* code using a single quoted functor as shown in the following example, where the first *term* is the type of each value stored in the *data*, and the second term is a base64 *string* containing the values:

```
data {

    ('data(byte,"BQz1OA=="));

}
```

When it comes to *unification*, a *data* can be unified with a *list*, as seen in the following knowledge:

```
data.print {

    ([])^       :-  true;
    ([:e|:r])   :-  console.puts(:e), ~self(:r);

}
```

```
?- #data(:D), #data.print(:D)
5
12
245
56
```

Subtitution is also supported as shown in the following example where we use a *knowledge* which build a *data term* holding as many random *numbers* as requested:

```
data.rnd {

    (1,:d)^          :-  daa.make(real32,[%rnd],:d);
    (:n,[%rnd|:d])  :-  sub(:n,1,:n1), ~self(:n1,:d);

}
```

```
?- #data.rnd(10,:D), daa.member(:v,:D)
-> ( 0.197688 ) := 1.00 (0.003) 1
-> ( 0.269155 ) := 1.00 (0.003) 2
-> ( 0.678092 ) := 1.00 (0.003) 3
-> ( 0.442051 ) := 1.00 (0.004) 4
-> ( 0.095426 ) := 1.00 (0.004) 5
-> ( 0.998170 ) := 1.00 (0.004) 6
-> ( 0.295657 ) := 1.00 (0.004) 7
-> ( 0.624670 ) := 1.00 (0.004) 8
-> ( 0.790462 ) := 1.00 (0.004) 9
-> ( 0.463402 ) := 1.00 (0.004) 10
```

Finally, see Section 5.5 on page 59 for details on the *primitives* that can be used with *data*.

3.4 Frame

In *fizz* , a *frame* is the equivalent of a *dictionary* in other languages. It stores key/value pairs. This is done by having a comma-separated collection of key/value pairs within curly braces. Here is an example:

```
gameboy.color {

    ({r = 0.509803, g = 0.784313, b = 0.294117});
    ({r = 0.325490, g = 0.670588, b = 0.392156});
    ({r = 0.164705, g = 0.549019, b = 0.349019});
    ({r = 0.000000, g = 0.294117, b = 0.282352});

}
```

While the value associated with a key can be any valid *term* (including a *Frame*), the key (also called *label*) can only be a valid *atom*. Unlike with *lists*, *unification* of two *frames* will only be done over the *labels* that both *terms* have in common.

There's a special kind of *frame* that can be used to *split* the content of a *frame* between specific values and the rest of the content. Its syntax makes use of the pipe character to split the frame between the pairs to unify (or substitue) and the rest of the pairs. For example:

```
?- set(:B,2), set(:F,{c = 3}), console.puts({a=1,b=:B|:F})
{c = 3, a = 1, b = 2}
-> (  ) := 1.00 (0.000) 1
```

Let's now consider the following knowledge, where we recursively iterate over some expected labels:

```
dump {
    cascade = yes
} {
```

```

    ({color = :v | :r}) :-  !is.variable(:v)^,
                            console.puts("color is ",:v),
                            ~self(:r);

    ({size = :v | :r})  :-  !is.variable(:v)^,
                            console.puts("size is ",:v),
                            ~self(:r);

    (_)                 :-  true;

}
```

```
?- #dump({size=small,a=4,color=red})
color is red
size is small
-> (  ) := 1.00 (0.002) 1
```

3.5 Functor

A *Functor* in *fizz* is akin to a *structure*, although it really is more of a *named list* (since a C-like structure will have fields). Here's an example where the likelihood of a given weather is given as a functor:

```
weather2 {

    (paris,rain(0.5),wind(0.1),sun(0.4),snow(0.1),fog(0.1));
    (london,rain(0.6),wind(0.1),sun(0.3),snow(0.0),fog(0.7));

}
```

When it comes to unifying *functors*. The *label* of each *functor* will be unified as well as each of the *terms*, therefore *arity* (the number of *terms*) of each *functors* also need to be the same.

3.6 Range

Range terms are a way to express a *range* of numerical values between *minimum* and *maximum* values. The syntax of a *range* is something that we have already encountered in Section 2.3 on page 4 when expressing the acceptable *truth value* range for a *predicate*. Here's an example where we look at the manufacturer-reported range of some electrical cars:

```
car.range {

    (ford(focus),76);
    (tesla(model_s),<210|315>);
    (tesla(model_x),<237|289>);
    (chevy(bolt),238);
    (nissan(leaf),107);

}
```

A *range* will unify with a fellow *range* but also with a *number* as long as it is within the *range*. If we were to query the above *knowledge* for a car with a range of at least 300 miles, we would do so like this: `@car.range(:x,300)` and get the variable `:x` bound to the value `tesla(model_s)`.

3.7 Regexp

A *Regexp* term is a way to represent a regular expression with which to unify *strings*. Such *terms* are specified in *fizz* code using a single quoted functor as shown in the following example:

```
?- rex.match(’regexp("(the|a)?\s?(dog|cat)\sis\s(wet|cold|sick)"),"cat is sick",:m)
-> ( ["cat is sick", "", "cat", "sick"] ) := 1.00 (0.001) 1
```

As *fizz* uses the PCRE2 library[1] to implement the regular expression support, the following flags (to be provided within a *list*) can be used to modify the way the expression is compiled:

`ANCHORED`	Force pattern anchoring
`ALLOW_EMPTY_CLASS`	Allow empty classes
`ALT_BSUX`	Alternative handling of \u, \U, and \x
`ALT_CIRCUMFLEX`	Alternative handling of în multiline mode
`ALT_VERBNAMES`	Process backslashes in verb names
`AUTO_CALLOUT`	Compile automatic callouts
`CASELESS`	Do caseless matching
`DOLLAR_ENDONLY`	$ not to match newline at end
`DOTALL`	. matches anything including NL
`DUPNAMES`	Allow duplicate names for subpatterns
`ENDANCHORED`	Pattern can match only at end of subject
`EXTENDED`	Ignore white space and comments
`FIRSTLINE`	Force matching to be before newline
`LITERAL`	Pattern characters are all literal
`MATCH_UNSET_BACKREF`	Match unset backreferences
`MULTILINE`	ând $ match newlines within data
`NEVER_BACKSLASH_C`	Lock out the use of \C in patterns
`NO_AUTO_CAPTURE`	Disable numbered capturing parentheses (named ones available)
`NO_AUTO_POSSESS`	Disable auto-possessification
`NO_DOTSTAR_ANCHOR`	Disable automatic anchoring for .*
`NO_START_OPTIMIZE`	Disable match-time start optimizations
`UNGREEDY`	Invert greediness of quantifiers
`USE_OFFSET_LIMIT`	Enable offset limit for unanchored matching

The `CASELESS` flag can be used to ignore the case during matching (note that flags are case-insensitive):

```
?- rex.match(’regexp("[a|b]+"),"aabb",:l)
-> ( ["aabb"] ) := 1.00 (0.001) 1
?- rex.match(’regexp("[a|b]+"),"ABABA",:l)
?- rex.match(’regexp("[a|b]+",[caseless]),"ABABA",:l)
-> ( ["ABABA"] ) := 1.00 (0.001) 1
```

Since, *regexp* are full fledged *terms*, they can be used in *predicates* and *prototype*’s entrypoint as shown in this example:

[1] see https://www.pcre.org/

```
str.is {

    ('regexp("[+-]?([0-9]*[.])?[0-9]+"),number)^    :- true;
    (_,string)                                      :- true;

}
```

Lastly, the *primitive* `rex.match` which we have used above can be used within a *contrained variable*. This allow the matching content to be accessed:

```
test {

    (:s?[rex.match('regexp("(the|a)?\s?(dog|cat)\sis\s(wet|cold|sick)"),:s,[_,_,_,:c])],:c) :- true;

}
```

```
?- #test("dog is sick",:l)
-> ( "sick" ) := 1.00 (0.002) 1
?- #test("dog is wet",:l)
-> ( "wet" ) := 1.00 (0.002) 1
?- #test("dog is gone",:l)
```

3.8 Variable

Variables in *fizz* , like in any *logic programming language*, are placeholders for any *terms*. As we have seen in several examples, the syntax for defining a *variable* is a *symbol* prefixed with a colon. Often when unification is happening, it is handy to indicate that we do not care about a given *term*. For such situations, we use the *wildcard variable*, which is a single underscore. If we take the `car.range` *knowledge* we defined above, we may want to list all the `tesla` cars, but without caring about the range of each model. We would express this in a predicate as follows: `@car.range(tesla(:m),_)`, and the `:m` *variable* will be bound to the values `model_s` and `model_x`.

Because inferences in *fizz* are distributed (within a single *substrate* or accross multiple networked *substrates*), the number of replies to a query need to be minimized whenever possible. As such, *variables* support *constraints* specifications. Let's look at an example where we are querying the `gameboy.color` knowledge we defined earlier:

```
?- @gameboy.color({r = :r, g = :g, b = :b })
-> ( 0.509803 , 0.784313 , 0.294117 ) := 1.00 (0.001) 1
-> ( 0.325490 , 0.670588 , 0.392156 ) := 1.00 (0.001) 2
-> ( 0.164705 , 0.549019 , 0.349019 ) := 1.00 (0.001) 3
-> ( 0 , 0.294117 , 0.282352 ) := 1.00 (0.001) 4
```

If we were only interested in the colors where the red component is within `0.1` and `0.4`, we could modify our query to use *primitives* to put constraints on the value bound to the `:r` *variables*:

```
?- @gameboy.color({r = :r, g = :g, b = :b }), gt(:r,0.1), lt(:r,0.4)
-> ( 0.325490 , 0.670588 , 0.392156 ) := 1.00 (0.001) 1
-> ( 0.164705 , 0.549019 , 0.349019 ) := 1.00 (0.001) 2
```

We now have two matching colors instead of four. However, we did that by filtering the answers we got to our query on the `gameboy.color` *knowledge*. By specifying *constraints* directly on the *variable* within the *predicate*, we could have only received the two matching *statements*:

```
?- @gameboy.color({r = :r?[gt(0.1),lt(0.4)], g = :g, b = :b})
-> ( 0.325490 , 0.670588 , 0.392156 ) := 1.00 (0.001) 1
-> ( 0.164705 , 0.549019 , 0.349019 ) := 1.00 (0.001) 2
```

Constraints are specified after a *variable* with a `question mark` followed by *list*, a *frame* or a *variable* (which will be bound at runtime to a *list* or *frame*). Each of the element in the *list* (which can be a *functor*, *range*, *symbol*, *lambda* or variable) is a constraint that any value bound to the *variable* must satisfy. In the above example, we indicated that the value for `:r` must be greater than `0.1` and less than `0.4`.

Constraints support multiple *functors* as listed in this table:

`if`	some *term* evaluate to *true*
`gt`	greater than
`gte`	greater than or equal
`lt`	lesser than
`lte`	lesser than or equal
`neq`	not equal
`aeq`	almost equal
`eq`	equal/unify
`eq.or.in`	value is equal to a term, or a list that contains a term
`neq.nor.in`	value is not equal to a term, nor a list that contains a term
`fun.label`	value is the label of a *functor*
`lst.member`	value is present in a *list*
`lst.except`	value is not present in a *list*
`lst.incl`	value is a list that include the items in a *list*
`lst.excl`	value is a list that exclude the items in a *list*
`is.atom`	value is an *atom term*
`is.binary`	value is a *binary term*
`is.string`	value is a *string term*
`is.symbol`	value is a *symbol term*
`is.number`	value is a *number term*
`is.regexp`	value is a *regexp term*
`is.guid`	value is a *guid term*
`is.list`	value is a *list term*
`is.range`	value is a *range term*
`is.frame`	value is a *frame term*
`is.func`	value is a *functor term*
`is.quirk`	value is a *quirk term*
`is.data`	value is a *data term*
`is.bound`	a value is bound
`is.unbound`	no value is bound yet
`is.even`	value is an even *number*
`is.odd`	value is an odd *number*
`is.final`	value is an unbound variable or (recursively) contains any unbound variable(s)
`str.find`	value is a *string* which contains a specified substring

Most *functors* requiere a single *term* except the `is.*` ones which can be given as a *symbol*, and `aeq` which expects two. *Constraints* can be use on any *variables*, including in a *prototype*'s entrypoint as shown here:

```
lst.zip {

    ([],[])^                          :- true;
    ([:e],[:e])^                      :- true;
    ([:e,:e|:r],:l)                   :- #lst.zip([:e|:r],:l);
    ([:e,:f?[neq(:e)]|:r],[:e|:l])    :- #lst.zip([:f|:r],:l);
```

```

}
```

Some of the *primitives* can be used directly as constraints. Check the specific details for a *primitive* to know if it supports this situation.

3.9 Constant

Constants in *fizz* are a special kind of *variable* whose content is static. Aside from the *constants* defined by the *runtime environment*, new ones can be defined via command line arguments. *Constants* do not support *constraints*, and are prefixed with a dollar sign. The following table lists all the *constants* provided by the *runtime environment*:

`$true`	the boolean value for *true*
`$false`	the boolean value for *false*
`$cores`	the number of CPU cores enabled for *fizz*
`$pi`	the numeral value of π
`$self`	when used within an *elemental*, it will be subtitued by its label.
`$guid`	when used within an *elemental*, it will be subtitued by its guid.
`$self.path`	when used within an *elemental*, it will be subtitued by the path of the file from which it was loaded from.

3.10 Volatile

Volatiles in *fizz* are a special kind of *constant* whose content is most likely to change in between *unifications*. They can be used to add, for example, a time stamp to a *statement* being asserted (added to a *knowledge*) like in this example:

```
?- assert(car(blue,%now))
-> (  ) := 1.00 (0.001) 1
?- @car(:color,:stamp)
-> ( blue , 1503602300.742353 )
```

The syntax for *volatiles* is similar to *constants*, but with a percent instead of the dollar sign. The following table lists all the *volatiles* currently supported:

`%now`	current time (UTC) in seconds since (Unix) Epoch
`%now.ms`	current time (UTC) in miliseconds since (Unix) Epoch
`%today`	date and time as a *string*
`%rnd`	a randomly generated *number* between 0 and 1
`%sym`	a randomly generated *symbol*
`%sym.3`	a randomly generated *symbol* of 3 characters length
`%sym.4`	a randomly generated *symbol* of 4 characters length
`%sym.6`	a randomly generated *symbol* of 6 characters length
`%sym.8`	a randomly generated *symbol* of 8 characters length
`%sym.10`	a randomly generated *symbol* of 10 characters length
`%sym.16`	a randomly generated *symbol* of 16 characters length
`%gui`	a randomly generated GUID as a *string*
`%seq`	a sequentially generated *symbol* of 10 characters length
`%uid`	a randonly generated UID *symbol* of 32 characters length

Because of their values are always changing, *volatiles* will always unify with anything. They should really not be used in a *statement*.

3.11 Quirk

Quirks in *fizz* can be understood as either *tuples* or annotated *terms*. They are composed of two *terms*, refererred as *head* and *tail*, separated by a caret. When such *term* is unified to any other *term*, it will be unified as whatever the *head term* is. Here's an example:

```
quirk {

    (:v?[lt(5)])^   :-  console.puts(:v," is less than five");
    (_^:v?[lt(5)])  :-  console.puts(:v," is less than five");

}
```

```
?- #quirk(2)
2 is less than five
-> (  ) := 1.00 (0.001) 1
?- #quirk(2^3)
2^3 is less than five
-> (  ) := 1.00 (0.001) 1
?- #quirk(6^3)
3 is less than five
-> (  ) := 1.00 (0.000) 1
```

3.12 Lambda

Lambda in *fizz* are limited bundle of imperative programming which unify or substitue to a single *term*. They are represented by a single *functor*, prefixed by a single quote. For example:

```
?- console.puts('add(4,5))
9
-> (  ) := 1.00 (0.000) 1
```

Any *terms* in the *functor* is accepted, including *lambda*:

```
?- set(:x,10), set(:y,'mul('add(:x,1),3))
-> ( 10 , 33 ) := 1.00 (0.000) 1
```

To be evaluated, a *lambda* must be one of the following supported functions:

`is.bound(expr)`	return *true* if *expr* is a bound *term*.
`is.atom(expr)`	return *true* if *expr* is an *atom*.
`is.number(expr)`	return *true* if *expr* is a *number*.
`is.string(expr)`	return *true* if *expr* is a *string*.
`is.symbol(expr)`	return *true* if *expr* is a *symbol*.
`is.binary(expr)`	return *true* if *expr* is a *binary*.
`is.list(expr)`	return *true* if *expr* is a *list*.
`is.func(expr)`	return *true* if *expr* is a *functor*.
`is.frame(expr)`	return *true* if *expr* is a *frame*.

`is.range(expr)`	return *true* if *expr* is a *range*.
`is.regexp(expr)`	return *true* if *expr* is a *regexp*.
`is.guid(expr)`	return *true* if *expr* is a *guid*.
`is.quirk(expr)`	return *true* if *expr* is a *quirk*.
`is.data(expr)`	return *true* if *expr* is a *data*.
`vars(expr)`	return a *list* of all the unbounded *variables* in *expr*.
`switch(expr,expr,expr))`	return the value associated (in a *frame* passed as second *term*) with the label (an *atom*) provided in the first *term*. If the label isn't found, the third *term* will be returned.
`eval(expr)`	return the result of the evaluation of *expr* as if was a *lambda*.
`add(expr,expr,...)`	return the addition of all arguments.
`sub(expr,expr,...)`	return the substraction of all arguments.
`mul(expr,expr,...)`	return the multiplication of all arguments.
`div(expr,expr,...)`	return the division of all arguments.
`div.int(expr,expr,...)`	return the integer division of all arguments.
`inv(expr)`	return the inverse of *expr*.
`max(expr,...)`	return the maximum value of all arguments.
`min(expr,...)`	return the minimum value of all arguments.
`mod(expr,expr)`	return the integer division between the arguments.
`sim(expr,expr)`	return the similarity between the two arguments.
`sum(expr,expr,...)`	return the sum of all arguments.
`inc(expr)`	return *expr* + 1.
`dec(expr)`	return *expr* - 1.
`abs(expr)`	return the absolute value of *expr*.
`ceil(expr)`	return the smallest integer value greater than or equal to *expr*.
`exp(expr)`	return *e* raised to the power of *expr*.
`floor(expr)`	return the largest integer value less than or equal to *expr*.
`log(expr)`	return the natural logarithm (base-e logarithm) of *expr*.
`log10(expr)`	return the common logarithm (base-10 logarithm) of *expr*.
`pow(expr,expr)`	return the first argument raised to the power of the second.
`round(expr)`	return the nearest integer value of *expr*.
`sign(expr)`	return the sign of *expr* (+1 or -1).
`sqrt(expr)`	return the square root of *expr*.
`atan2(expr,expr)`	return the principal value of the arc tangent of the first argument divided by the second (expressed in degrees).
`cos(expr)`	return the cosine of the angle given with *expr* (in degrees).
`acos(expr)`	return the arc-cosine of *expr* (in degrees).
`sin(expr)`	return the sine of the angle given with *expr* (in degrees).
`asin(expr)`	return the arc-sine of *expr* (in degrees).
`d2r(expr)`	return the conversion of expr from degree to radian.
`r2d(expr)`	return the conversion of expr from radian to degree.
`if(expr,expr,expr)`	return the second argument if the first argument is true, otherwise the third argument.
`eq(expr,expr)`	return true if the arguments are equal, false otherwise.
`eq.any(expr,...)`	return true if the first argument is equal to any of the following, false otherwise.
`neq(expr,expr)`	return false if the arguments are equal, true otherwise.
`gt(expr,expr)`	return true if the first argument is greater than the second.
`gte(expr,expr)`	return true if the first argument is greater or equal to the second.
`lt(expr,expr)`	return true if the first argument is lesser than the second.
`lte(expr,expr)`	return true if the first argument is lesser or equal to the second.
`and(expr,...)`	return the boolean `AND` of all arguments.
`or(expr,...)`	return the boolean `OR` of all arguments.
`not(expr)`	return the boolean `NOT` of the argument.
`xor(expr,...)`	return the boolean *exclusive disjunction* of all arguments.
`any(expr,...)`	return the first of the arguments that isn't an unbounded variable.
`uny(expr,expr)`	return true if both expression can be unified.
`str.cat(expr,...)`	return the concatenation of all the arguments (as a string).
`str.length(expr)`	return the length of the argument (a string).
`sym.cat(expr,...)`	return the concatenation of all the arguments (as a symbol).
`sym.length(expr)`	return the length of the argument (a symbol).

`lst.append(expr,expr)`	return a new list where the second argument is appended to a list (the first argument).
`lst.prepend(expr,expr)`	return a new list where the second argument is added to the head o a list (the first argument).
`lst.item(expr,expr)`	return a given item (second argument) from a list (first argument).
`lst.length(expr)`	return the length of the list given as *expr*.
`lst.head(expr)`	return the first item in the *list* given as *expr*. Return *expr* if it's not a list.
`lst.tail(expr)`	return the last item in the *list* given as *expr*. Return *expr* if it's not a list.
`lst.rest(expr)`	return list given as *expr* minus its first item. Return *expr* if it's not a list.
`lst.bulk(expr)`	return list given as *expr* minus its last item. Return *expr* if it's not a list.
`lst.find(expr,expr)`	return a list of indices that point to all the items in the list (first argument) that unifies with the second term.
`lst.member(expr,expr)`	return true if the second argument is present in the first argument (a list).
`lst.excl(expr,expr)`	return true if the second argument is not present in the first argument (a list).
`lst.merge(expr,...)`	return a *list* from all the arguments. When an argument is a *list*, its content will be placed into the returned *list*.
`frm.label(expr,expr)`	test if a given label (the second *term*) is present in the first (a *frame*).
`frm.store(expr,expr,expr)`	return a copy of then first *term* (a *frame*) with a value (the third *term* associated with the label (the second *term*) stored in it.
`frm.fetch(expr,expr,expr)`	return the value associated with the label (the second *term*) in a frame (first *term*.) If the label isn't present, the third *term* will be returned.
`frm.merge(expr,...)`	return a *frame* which contains the merging of all the *frame* arguments passed to it.If a label already exists, its value will be replaced.
`frm.erase(expr,expr)`	return a copy of the first *term* (a *frame*) in which a value was removed using the second *term* (an *atom*) as the key to be removed. If the second *term* is a *list*, all *atoms* it hold will be considered as key to be removed.
`frm.pairs(expr)`	return a *list* of all the label/value pairs in the *term* (a *frame*). Each of the pairs will be stored in a *list* of two elements.
`qrk.head(expr)`	return the first item in the *quirk* given as *expr*.
`qrk.tail(expr)`	return the last item in the *quirk* given as *expr*.
`fun.label(expr)`	return the label of a *functor* given as *expr*.
`fun.terms(expr)`	return the terms of a *functor* given as *expr*.
`fun.make(expr,expr)`	return a functor build from a label (the first *term*) and a *list* of *terms* (second *term*).

4 Console

4.1 Usage

Because of its *asynchronous* and *concurrent* nature, *fizz* provides a *console* with a slightly unusual mode of operation. The default state of the *console* is to display any outputs coming from the *runtime* or from the queries entered by the user. Here's the *console* when the program is started:

```
$ ./fizz.x64
fizz 0.1.0-P (20171116.1221) [x64|3]
```

To switch to input, for example to enter a query or any of the supported *console*'s command, press the `ESC` key or one of the `arrow` keys. When the *console* is waiting for user input, it will display a `?-`. If `Ctrl-C` is pressed, the *console* will exit the input state. The `up` and `down` arrow keys also serve to cycle thru the history. While, the *console* is in such mode, any output coming from the *runtime* will be buffered until the mode is exited. Press the `enter` key to exit the *input* mode. If a query or command was entered, it will be executed (in most case asynchronously) and any result will be printed:

```
fizz 0.1.0-P (20171116.1221) [x64|3]

load : loading manual.fizz ...
```

```
load : loaded  manual.fizz in 0.013s

?- @gameboy.color(:color)
-> ( {r = 0.509803, g = 0.784313, b = 0.294117} ) := 1.00 (0.001) 1
-> ( {r = 0.325490, g = 0.670588, b = 0.392156} ) := 1.00 (0.001) 2
-> ( {r = 0.164705, g = 0.549019, b = 0.349019} ) := 1.00 (0.001) 3
-> ( {r = 0, g = 0.294117, b = 0.282352} ) := 1.00 (0.001) 4
```

Each solution to a query will be presented as a *statement* where each *variable* becomes one of the *statement*'s terms (in the order they appears in the predicates). The *truth value* will be printed after, followed by the elapsed time (in seconds) since the query was sent. The last number is a sequential number for the *reply*. It is worth noting that in *fizz* a query will not be stopped at the first answer.

If you know that a *query* is going to generate many *replies* that you don't care about, you can set the property `verbose` of the console to `no` using the `poke` *primitive*. For example:

```
?- /spy(append,gameboy.color)
spy : observing gameboy.color
?- poke(verbose,no)
?- @gameboy.color(:color)
spy : [1589955674.084] Q @gameboy.color(:color) (14.999913)
spy : [1589955674.084] R gameboy.color({r = 0.509803, g = 0.784313, b = 0.294117}) (14.999781)
spy : [1589955674.084] R gameboy.color({r = 0.325490, g = 0.670588, b = 0.392156}) (14.999781)
spy : [1589955674.084] R gameboy.color({r = 0.164705, g = 0.549019, b = 0.349019}) (14.999781)
spy : [1589955674.084] R gameboy.color({r = 0, g = 0.294117, b = 0.282352}) (14.999781)
```

Another way to silence some of the response is to replace the label of given *variable* that you do not wish to see the possible values by an all upper case label. This is a convention that only works within the console. For example:

```
?- #product(:n,:m,:Y), gt(:Y,2003)
-> ( model_e , tesla ) := 1.00 (0.001) 1
-> ( iphone_x , apple ) := 1.00 (0.001) 2
-> ( vive , htc ) := 1.00 (0.001) 3
-> ( iphone , apple ) := 1.00 (0.001) 4
-> ( iphone_3GS , apple ) := 1.00 (0.001) 5
-> ( 7710 , nokia ) := 0.90 (0.001) 6
```

When invoking the *executable*, the arguments of the command line can be any numbers of strings specifying the path and name of files to be loaded by the *runtime*, as seen in the above example. If the path leads to a folder, it will be assumed that it is a previously frozen *runtime* enviroment to kindle. The command line option `-l` can be used to switch the console logging on. This option will expect as argument the path and name of the log file to be created. For example:

```
$ ./fizz.x64 -l test.log manual.fizz
```

The command line option `-q` can be used to specify a query to be executed right after the executable enter its Read–Eval–Print Loop (REPL). Be aware, thought that loading files in *fizz* is done asynchronously. Therefore a query using any yet-to-be loaded *knowledges* will fail. For example:

```
./fizz.x64 -q "/load(\"manual.fizz\")"
fizz 0.1.0-P (20171116.1221) [x64|3]
```

```
?- /load("manual.fizz")
load : loading manual.fizz ...
load : loaded  manual.fizz in 0.013s
```

Any key pressed while outside of the console input state will cause a `console.keypress` *statement* to be broadcasted in the *substrate*. Any *elemental* can make use of it (via an activable *predicate*) and execute inferences based on the key that was pressed. The sole *term* of that *statement* is the ASCII code of the key. As an example, here's a *knowledge* which display an hint to the user each time it press a key:

```
help {

    () :- @console.keypress(_), hush, console.puts("press ESC to enter input mode");

}
```

Lastly, pressing `Ctrl-C` outside of the input state, will cause the *executable* to terminate.

4.2 Adjusting the *runtime*

Severals parameters of the *runtime environment* can be adjusted by creating (or modifying) a `JSON` file. In order for the executable to use that file when it starts, the file must have the same name as the executable and have the exension `.json`. Here's an example of a file that adjusts all the possible parameters:

```
{
    "runtime" : {
        "scheduler" : {
            "threads"  : 4,
            "affinity" : true,
            "spinning" : 4
        },
        "offloader" : {
            "minpool"  : 1,
            "maxpool"  : 4,
            "timeout"  : 750,
            "affinity" : false
        },
        "livereload" : {
            "enabled"   : true,
            "interval"  : 250
        }
    },
    "substrate" : {
        "ttl" : {
            "type" : "real",
            "data" : 55.0
        },
        "grace" : {
            "type" : "real",
            "data" : 0.5
        },
        "sspr" : {
            "type" : "uint",
            "data" : 8
        },
```

```
        "pulse" : {
            "type" : "uint",
            "data" : 250
        },
        "epsilon" : {
            "type" : "real",
            "data" : 0.000001
        },
        "lettered" : {
            "type"  : "string",
            "data"  : "MRKCBFSolver"
        },
        "bundle.len" : {
            "type" : "uint",
            "data" : 1024
        },
        "bundle.tmo" : {
            "type" : "real",
            "data" : 0.5
        },
        "mzttl" : {
            "type" : "real",
            "data" : 1.5
        },
    },
    "modules" : {
        "www" : {
            "maxrequest" : 4,
            "maxcontent" : 2048
        }
    }
}
```

It contains three sections: the `runtime`, `substrate` and `modules`. The former adjusts the threading and multi-cores models of the *runtime* while `substrate` adjusts the common behavior of all *elemental* objects will use. The later provides parameters for the modules that may be loaded.

Let's look at the key/value pairs in the `scheduler` section:

`threads`	represents the number of threads to be used per available core. This number will not change at any point in time
`affinity`	if set to `true`, each thread will be assigned to a given core of the host
`spinning`	the maximum number of consecutive time an *elemental* will get time on a core before it gets swapped out for another *elemental*. The lesser the value the more the *scheduler* will *round-robin* between the *elementals*.

The `offloader` section is responsible for tuning the part of the runtime that handles *offloaded* processing using a dynamically resizable thread pool. The execution of any *primitives* flagged as *offloaded* will be executed on the pool instead of being executed within the *elemental* object calling it. The key/value pairs meanings is as follows:

`minpool`	the minimum number of threads in the pool at any given time.
`maxpool`	the maximum number of threads in the pool at any given time.
`timeout`	the maximum amount of time a non-busy thread will wait before it exits the pool.
`affinity`	if set to `true`, each thread will be assigned to a given core of the host.

The `livereload` section deals with the automatic *live code reload* built in *fizz* . If this section is not present in the configuration file, this functionality will not be available. The command line option `-n` can be used to force this functionality to be disabled even if it is enabled in the configuration `JSON` file. The key/value pairs meanings is as follows:

`enabled`	`true` to enable functionality, `false` to disable.
`interval`	interval of time (in ms) in between checks of the loaded scripts file's timestamp.

Because the `substrate` section of the JSON file deals with the configuration of each *elemental*, the format that is expected is a little different. The meaning of each value is:

`ttl`	this is the default *time to live* for anything posted on the *substrate* (in seconds).
`grace`	this is the *grace period* for any *query* (in seconds).
`sspr`	the maximum number of *statements* to be included in a single *query* reply. If there are more *statements* to be sent, more replies will be sent.
`pulse`	the frequency (in miliseconds) at which each *elementals* gets to perform cleanups and other cyclic tasks. The lower the value, the more CPU will be used.
`epsilon`	the upper bound on the relative error due to rounding in floating point arithmetic to be used when comparing *numbers*.
`lettered`	default *elemental* class to be used when creating *elemental* to handle asserted *statements*.
`bundle.len`	the maximum number of *statement* that can be bundled into a single *knowledge* before it is *asserted* in the *substrate*.
`bundle.tmo`	the timeout value (seconds) before bundled *statements* are to be asserted if no other *statements* is added to the bundle.
`mzttl`	this is the *time to live* for any statements that is cached by an *elemental* set to *memoize* (in seconds).

If a query is going to take longer than the default `ttl`, an alternate value can be specified for a *predicate* by providing its properties. For example:

```
?- #some_long_query(:a,:b) {ttl=25}
```

The `httpclient` section (in the `www` section of `modules`) is responsible for tuning the built-in HTTP client used by, for example, the *elemental* class `FZZCWebAPIGetter`. The key/value pairs meanings is as follows:

`maxrequest`	maximum number of concurrent request (0 for no limit).
`maxcontent`	maximum size of the content to store in RAM, before storing it into a temporary file.

Lastly, there are two command line options of interest: `-s` and `-c`. The formost can be used to specify an alternate settings `JSON` file as show here:

```
./fizz.x64 -s laptop.json manual.fizz
fizz 0.1.0-P (20171116.1221) [x64|3]

load : loading manual.fizz ...
load : loaded  manual.fizz in 0.013s
```

The latter allows *constants* to be defined as shown in this example:

```
./fizz.x64 -c user=$USER
fizz 0.1.0-P (20171116.1221) [x64|3]

?- console.puts($user)
jlv
-> (  ) := 1.00 (0.000) 1
```

The expected syntax for each defined constants is `label=value`. The value can be any *term* while the label is expected to be a *symbol*. Multiple `-c` options can be given.

4.3 Solution

A *solution* is a `JSON` file that can be loaded by *fizz* and describe a given set of source files, global constants and modules to be loaded. Here's an example of such file for the `linkg.fizz` sample:

```
{
    "solution" : {
        "modules" :   ["modLGR"],
        "sources" :   ["linkg.fizz"],
        "globals" :   [],
        "queries" :   []
    }
}
```

To be valid, such file much constains a `solution` object, itself containing the following (all optional) labels:

- `modules` a list of modules to be loaded (without file extensions)
- `sources` a list of sources to be loaded (which path is relative to the path of the *solution* file). When one of the element in the array is itself an array, the files listed in it will be loaded sequentially instead of concurrently.
- `globals` a list of objects describing the *constants* to be created. Each of the objects must contains two label/value pairs: `label` and `value`.
- `queries` a list of queries (in the form of strings containing *predicates*) to be executed once all the sources and modules files have been loaded.

Here's an example of the solution file for the *weather.fizz* sample:

```
{
    "solution" : {
        "modules" :   [],
        "sources" :   [
            "weather.fizz"
        ],
        "globals" :  [
            {
                "label" : "api.key",
                "value" : "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
            }
        ],
        "queries" : []
    }
}
```

To use this *solution* you will need to replace the value for `api.key` by your own key.

4.4 Commands

Commands differs from *queries* by starting with a slash. Otherwise, their syntax is similar to a *predicate* (minus the *truth value* range). For example:

```
?- /load("./samples/manual.fizz")
load : loading ./samples/manual.fizz ...
load : loaded  ./samples/manual.fizz in 0.011s
```

Will load the contents of the `manual.fizz` file into the *runtime*.

bye

`/bye`

Close the console and terminate the executable.

create

`/create(`*symbol*`,`*symbol*`,`*frame*`,`*number?*`)`

Creates one (or more if a fourth *terms* is provided) *elemental* object which *label* will be the first *term*. The second *term* is the name of the /em class on which the *elemental* should be based. The third *term* contains the *properties* of the object. For example, to create ten *elementals* labeled `product` each with a *statements* limit of `1000`, we would type:

```
?- /create(product,MRKCLettered,{s.limit = 1000},10)
create : okay.
```

cpus

`/cpus`

Print to the *console* the number of cores the host computer has. This can be handy when you do not know that answer and want to adjust the configuration of the *runtime*.

```
?- /cpus
host has 4 CPUs
```

delete

`/delete(`*symbol*`|`*string*,*symbol*`|`*string**`)`

The `delete` command allows for *elementals* to be removed from the *substrate*. The command will accept any numbers of *symbols* or *string* as its *terms*. The only supported *strings* are GUID while the *symbols* can be either an *alias* or a *knowledge*'s label. When the later is used, all *elementals* objects with this label will be removed:

```
?- /delete(number,fill.it,"3716b075-7d64-2440-eda0-96b1b3e9ae20")
delete : completed in 0.000s
```

If any of the *terms* doesn't resolve into an actual *elemental*, the command will still complete successfully.

export.csv

`/export.csv(`*string*`,`*functor*`,`*string*`,`*list*`,`*frame?*`)`

This command exports *statements* into a file storing tabular data (*numbers* and *strings*) in a plain text format, using the character from the third *term* as delimiter for the generated lines. The first *term* indicates the path and filename of the file to be created, while the second *term* is the *predicate* to be queried for. The *list* provided as fourth *term* contains the index of each columns (starting from 0) to be included in each lines. If provided, the fifth *term* is a *frame* which can specify a timeout value (in seconds) after which the command shall complete (with the label `tmo`); and if the *truth value* of each *statements* is to be added as a column (with the label `truth`). When no timeout is provided, the default is half a second.

As an example, let's consider the following two *knowledges*:

```
product {

    (model_e,tesla,2012);
    (iphone_x,apple,2018);
    (vive,htc,2015);
    (coconut_water,zico,2000);

}

product {

    (iphone,apple,2007);
    (iphone_3GS,apple,2009);
    (7710,nokia,2005) := 0.9;

}
```

To export all *statements* with a third term greater than `2005`, we would use the command as follow:

```
?- /export.csv("products.csv",product(_,_,_?[gt(2005)]),",",[0,2],{truth = yes})
export.csv : wrote 5 lines in 0.021s.
```

Which will generate a `products.csv` file containing:

```
iphone,2007,1.0
iphone_3GS,2009,1.0
model_e,2012,1.0
iphone_x,2018,1.0
vive,2015,1.0
```

By using an intermediate *knowledge* instead of directly querying the *knowledge* that interests us, we could have further filter and/or modify the *statements* generated. Here's a simple example which add a GUID to each of the lines that will be stored in the CSV file:

```
product.g {

    (:l,:m,:y,%gui) :- #product(:l,:m,:y?[gt(2005)]);

}
```

The `export.csv` command will then be:

```
?- /export.csv("products.csv",product.g(_,_,_,_),",",[])
export.csv : wrote 5 lines in 0.016s.
```

And it the CSV file contents will be:

```
model_e,tesla,2012,3c5b83d9-278e-654a-3c88-07d99d2c1fd0
iphone_x,apple,2018,5036ef91-7a5f-904b-fa89-771e852f492e
vive,htc,2015,9369d034-941b-de47-66b8-877da629fae5
iphone,apple,2007,6fa0953c-f6b4-bd45-8bb7-6e21ab9df9e8
iphone_3GS,apple,2009,33118137-4253-0241-82ba-951a3ed16de9
```

export.json

`/export.json(`*string*,*functor*,*frame?*`)`

This command exports *statements* into a JSON file. The first *term* indicates the path and filename of the file to be created, while the second *term* is the *predicate* to be queried for. If provided, the third *term* is a *frame* which can specify a timeout value (in seconds) after which the command shall complete (with the label `tmo`). When no timeout is provided, the default is half a second. Note that only *string*, *number*, *list* and *frame* can be exported to JSON.

As an example, let's consider the following `gameboy.color` *knowledge*:

```
gameboy.color {

    ({r = 0.509803, g = 0.784313, b = 0.294117});
    ({r = 0.325490, g = 0.670588, b = 0.392156});
    ({r = 0.164705, g = 0.549019, b = 0.349019});
    ({r = 0.000000, g = 0.294117, b = 0.282352});

}
```

If we wanted to export the colors for which the *red value* if in between `0.1` and `0.4`, we would do:

```
?- /export.json("color.json",gameboy.color({r = \_?[gt(0.1),lt(0.4)]}))
export.json : wrote file color.json
```

And the generated JSON file will contain:

```
{
    "gameboy.color" : [ {
        "r" : 0.325490,
        "g" : 0.670588,
        "b" : 0.392156
    } , {
        "r" : 0.164705,
        "g" : 0.549019,
        "b" : 0.349019
    } ]
}
```

Since there was more than one matching *statement*, the generated JSON object will contain an array with all the *frames* that were in the *statements*. The key for that array will be the label of the *functor* used to query the *substrate*. If the array only contains a single *frame*, the *frame* only will be exported as we can see in the generated file:

```
{
    "r" : 0.325490,
    "g" : 0.670588,
    "b" : 0.392156
}
```

When the *statements* to be exported do not contains a single *term*, all the exportable *terms* will be exported within a JSON array. For example, if we consider the following *knowledge*:

```
product {

    (model_e,tesla,2012);
    (iphone_x,apple,2018);
    (vive,htc,2015);
    (coconut_water,zico,2000);

}

product {

    (iphone,apple,2007);
    (iphone_3GS,apple,2009);
    (7710,nokia,2005) := 0.9;

}
```

and export it as follow:

```
?- /export.json("products.json",product(_,_,_?[gt(2005)]))
export.json : wrote file products.json
```

The JSON file will then contains:

```
{
    "product" : [ [ "iphone" , "apple" , 2007 ]  ,
                  [ "iphone_3GS" , "apple" , 2009 ]  ,
                  [ "model_e" , "tesla" , 2012 ]  ,
                  [ "iphone_x" , "apple" , 2018 ]  ,
                  [ "vive" , "htc" , 2015 ]
                ]
}
```

freeze

`/freeze(`*string*`)`

This command *freezes* the *runtime* enviroment to a binary format that can be kindled at a later point. The only accepted *term* is the path of the folder in which the saving to be done. Please note that any on-going *query* is not preserved.

history.cls

`/history.cls`

Clear the *console*'s history.

history.len

`/history.len(`*number*`)`

Change the length of the *console*'s history. The default is `100`.

import.csv

`/import.csv(`*string*`,`*symbol*`,`*string*`,`*list*`,`*number?*`,`*number?*`)`

Imports data from a file storing tabular data (*numbers* and *strings*) in a plain text format (using any characters from the third *term* as delimiter and generates *statements* from each line. The first *term* indicates the path and filename of the file to be imported, while the second *term* is the label to be used for the *statements* that will be generated. The *list* provided as fourth *term* contains the number of each columns (starting from 0) to be extracted from each line of the file and put in the *statement*. If provided, the fifth *term* is the number of lines from the file to skip and if there is a fifth *term* it will be the number of lines to be processed.

If we wanted to import a `CSV` file such as this:

```
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
7.0,3.2,4.7,1.4,Iris-versicolor
6.4,3.2,4.5,1.5,Iris-versicolor
6.3,3.3,6.0,2.5,Iris-virginica
5.8,2.7,5.1,1.9,Iris-virginica
```

We would do as follows:

```
?- /spy(append,iris)
spy : observing iris
?- /import.csv("iris.data",iris,",",[])
import.csv : 6 lines read in 0.001s.
spy : S iris(5.100000, 3.500000, 1.400000, 0.200000, "Iris-setosa") := 1.00
spy : S iris(4.900000, 3, 1.400000, 0.200000, "Iris-setosa") := 1.00
spy : S iris(7, 3.200000, 4.700000, 1.400000, "Iris-versicolor") := 1.00
spy : S iris(6.400000, 3.200000, 4.500000, 1.500000, "Iris-versicolor") := 1.00
spy : S iris(6.300000, 3.300000, 6, 2.500000, "Iris-virginica") := 1.00
spy : S iris(5.800000, 2.700000, 5.100000, 1.900000, "Iris-virginica") := 1.00
```

Since we wanted all the columns to be used, we simply provide an empty *list* as the fourth *term*. Also, if a column is detected as holding a numerical value, it will be automatically converted as a *number*. If we had wanted to convert the last column into a *symbol* (instead of the *string* we are getting), we would have had to use an intermediary *elemental* object which would have made the conversion. Something such as this:

```
convert {

    ()  :-  @input(:e1,:e2,:e3,:e4,:l),
            str.tolower(:l,:l1),str.tosym(:l1,:l2),
```

```
            assert(iris(:e1,:e2,:e3,:e4,:l2),1.0f);

}
```

It simply states that each time an `input` *statement* is broadcasted in the *substrate* (which is what `import` does), the last *term* will be converted to a *symbol* after having its case changed to lowercase. Finally, a new `iris` *statement* is asserted. Running it we now get:

```
?- /spy(append,iris)
spy : observing iris
?- /import.csv("iris.data",input,",",[])
import.csv : 6 lines read in 0.001s.
spy : S iris(5.100000, 3.500000, 1.400000, 0.200000, iris-setosa) := 1.00
spy : S iris(4.900000, 3, 1.400000, 0.200000, iris-setosa) := 1.00
spy : S iris(7, 3.200000, 4.700000, 1.400000, iris-versicolor) := 1.00
spy : S iris(6.400000, 3.200000, 4.500000, 1.500000, iris-versicolor) := 1.00
spy : S iris(6.300000, 3.300000, 6, 2.500000, iris-virginica) := 1.00
spy : S iris(5.800000, 2.700000, 5.100000, 1.900000, iris-virginica) := 1.00
```

import.json

`/import.json(`*string*`,`*symbol*`,`*list?*`)`

Imports data from a JSON file. The first *term* indicates the path and filename of the file to be imported, while the second *term* is the label to be used for the *statement* that will be generated. If provided, the third *term* is a list of options to be used for the processing of the JSON objects contained in the file: `stringify` will keep all strings as *string terms*, `symbolize` will force all strings to be converted as *symbols*. The default behavior is to convert the strings that can be considered *symbol* as such.

As example, let's look at importing the *foreign exchange rates* from such a site as fixer.io[2]. For the sake of simplicity, the JSON file below was abbreviated:

```
{
    "base":"USD",
    "date":"2017-12-08",
    "rates":{
            "AUD":1.3303,
            "BGN":1.6656,
            "BRL":3.2733,
            "CAD":1.2836,
            "CHF":0.99676,
            "CNY":6.6197,
            "CZK":21.764,
            "DKK":6.3377,
            "GBP":0.7454
    }
}
```

When we import the file, it will generate a *statement* containing a single *frame*. To further process the *frame* to fit your need, you will need to use some supporting *knowledge*, so that the right *statements* can be generated. In the sample `etc/samples/fixer.fizz` you will find such support code that will process the JSON data from above:

[2] `http://api.fixer.io/latest?base=USD`

```
?- /spy(append,conversion)
spy : observing conversion
?- /import.json("./etc/usd-mini.json",input)
import.json : ./etc/usd-mini.json read in 0.001s.
spy : S conversion(USD, AUD, 1.330300) := 1.00 (700.000000)
spy : S conversion(USD, BGN, 1.665600) := 1.00 (700.000000)
spy : S conversion(USD, BRL, 3.273300) := 1.00 (700.000000)
spy : S conversion(USD, CAD, 1.283600) := 1.00 (700.000000)
spy : S conversion(USD, CHF, 0.996760) := 1.00 (700.000000)
spy : S conversion(USD, CNY, 6.619700) := 1.00 (700.000000)
spy : S conversion(USD, CZK, 21.764000) := 1.00 (700.000000)
spy : S conversion(USD, DKK, 6.337700) := 1.00 (700.000000)
spy : S conversion(USD, GBP, 0.745400) := 1.00 (700.000000)
```

The code in `fixer.fizz` splits the work over two *elementals*: `process` and `process.rates`:

```
process {

    () :-   @input(:f),
            frm.fetch(:f,base,:base),
            frm.fetch(:f,rates,:r),
            #process.rates(:base,:r);

}
```

The first one, activated when an `input` *statement* is published on the *substrate*, fetchs from the *frame* it contains the value for the `base` and `rates` labels and pass them to the second *elemental*:

```
process.rates {

    (:base,:f) :-   frm.fetch(:f,:l?[is.symbol],:v?[is.number]),
                    assert(conversion(:base,:l,:v),1.0f);

}
```

Since the `rates` are contained in a single *frame*, the *elemental*, concurrently fetchs all the label/value pairs from it, checking that they both match the expected type, then a new `conversion` statement is asserted.

import.txt

`/import.txt(`*string*, *symbol*, *number?*, *number?*`)`

Imports data from a file storing data in plain text and generates a single *statements* from each line. The first *term* indicates the path and filename of the file to be imported, while the second *term* is the label to be used for the *statements* that will be generated. If provided, the third *term* is the number of lines from the file to skip and if there is a fourth *term* it will be the number of lines to be processed. Each of the *statement* will have two *terms*: the first being a sequential number (starting at 0) and the second a *string* containing the whole line:

```
?- /spy(append,dna)
spy : observing dna
?- /import.txt("./etc/data/U00096.3.txt",dna,1,10)
spy : S dna(0, "AGCTTTTCATTCTGACTGCAACGGGCAATA...AAAAAAGAGTGTCTGATAGCAGCTTCTG") := 1.00
    (700.000000)
```

```
spy : S dna(1, "AACTGGTTACCTGCCGTGAGTAAATTAAAA...ACTAAATACTTTAACCAATATAGGCATA") := 1.00
    (700.000000)
spy : S dna(2, "GCGCACAGACAGATAAAAATTACAGAGTAC...CATTAGCACCACCATTACCACCACCATC") := 1.00
    (700.000000)
spy : S dna(3, "ACCATTACCACAGGTAACGGTGCGGGCTGA...GAAAAAAGCCCGCACCTGACAGTGCGGG") := 1.00
    (700.000000)
spy : S dna(4, "CTTTTTTTTTCGACCAAAGGTAACGAGGTA...GAAGTTCGGCGGTACATCAGTGGCAAAT") := 1.00
    (700.000000)
spy : S dna(5, "GCAGAACGTTTTCTGCGTGTTGCCGATATT...GCAGGGGCAGGTGGCCACCGTCCTCTCT") := 1.00
    (700.000000)
spy : S dna(6, "GCCCCCGCCAAAATCACCAACCACCTGGTG...CATTAGCGGCCAGGATGCTTTACCCAAT") := 1.00
    (700.000000)
spy : S dna(7, "ATCAGCGATGCCGAACGTATTTTTGCCGAA...CGCCGCCCAGCCGGGGTTCCCGCTGGCG") := 1.00
    (700.000000)
spy : S dna(8, "CAATTGAAAACTTTCGTCGATCAGGAATTT...CCTGCATGGCATTAGTTTGTTGGGGCAG") := 1.00
    (700.000000)
spy : S dna(9, "TGCCCGGATAGCATCAACGCTGCGCTGATT...GTCGATCGCCATTATGGCCGGCGTATTA") := 1.00
    (700.000000)
import.txt : 10 lines read in 0.001s.
```

kindle

`/kindle(`*string*`)`

This command loads a *runtime* enviroment from a previously saved binary format. The only accepted *term* is the path of the folder in which the saving was done. Using `kindle` and `freeze` are more efficient than `load` and `save` since it use a direct binary format instead of an intermediary text format that would need to be parsed. However, it is not possible to edit the *knowledge* with a text editor.

knows

`/knows(`*symbol*`|`*string*`|`*guid*`)`

Check if an *elemental* object is present on the *runtime* using its alias (when the argument is a *symbol*) or its GUID (when the argument is a *string* or a *guid*). In the following example, we modify the `car.range` *knowledge* to specify an alias for the *elemental* object that will get created:

```
car.range {

    alias = crange

} {

    (ford(focus),76);
    (tesla(model_s),<210|315>);
    (tesla(model_x),<237|289>);
    (chevy(bolt),238);
    (nissan(leaf),107);

}
```

We can then use that alias with the `/knows` command:

```
?- /knows(c.range)
no
?- /knows(crange)
yes
```

list

`/list`

This command generates a list of all the *elemental* objects presents on the *substrate*. Each of the output lines, will contains, in order, the GUID, the class, label and, if available, the alias of each *elementals*:

```
?- /list
list : 288a77db-bab2-1748-38af-892fcf18d112 MRKCLettered        blobs
list : bf006e31-4bd4-c348-a1a7-0449fb0a167f MRKCLettered        car.range       (crange)
list : 1bb328bb-4938-8a43-9db0-2a1685acc19b MRKCLettered        color
list : 3cfc2da3-8728-0d49-22a0-761d19af28bb MRKCLettered        gameboy.color
list : c9928201-4dbd-5e4d-bab7-ee9e13c771dc MRKCLettered        identifiers
list : 47d3366d-5794-0949-d7a4-f7e462dfaa24 MRKCBFSolver        lst.print
list : 966b0df2-7010-6542-5f83-5cedb64afadb MRKCBFSolver        maybe_rainbow
list : 4048e8be-8adc-0b4a-b880-4968dbaff277 MRKCBFSolver        multiplier
list : 9a5d0527-34d8-ee44-3f9d-7a8522d51cc0 MRKCLettered        product
list : 46d90c88-339d-fa40-da96-3cf068763eca MRKCLettered        product
list : 87240913-e8a1-9e43-3a9c-4b9f47e15b27 MRKCBFSolver        product.g
list : aa7f7a44-d894-c54d-db96-c537d7fb117c MRKCLettered        quotes
list : cfff6ad5-17ce-db43-3d8b-9855d8001539 MRKCRandomizer      rand
list : dd875f1a-9596-a649-7fbb-09420e20396f MRKCBFSolver        surely_raining
list : 6d4e5104-22a2-ee43-3eb3-073f45b08a1e MRKCLettered        weather
list : cb6a0d33-0000-0644-f89a-c7e678060aff MRKCLettered        weather2
list : 45f63bd5-b824-594f-e990-1487247ef64d MRKCLettered        yearly_stats
list : 17 elementals listed in 0.000s
```

load

`/load(`*string+*`)`

The `load` command allows *knowledge* to be loaded from (properly formatted) text files. All terms in the `predicate` are expected to be *strings*.

```
?- /load("./samples/manual.fizz")
load : loading ./samples/manual.fizz ...
load : loaded  ./samples/manual.fizz in 0.011s
?- @gameboy.color(:color)
-> ( {r = 0.509803, g = 0.784313, b = 0.294117} ) := 1.00 (0.001) 1
-> ( {r = 0.325490, g = 0.670588, b = 0.392156} ) := 1.00 (0.001) 2
-> ( {r = 0.164705, g = 0.549019, b = 0.349019} ) := 1.00 (0.001) 3
-> ( {r = 0, g = 0.294117, b = 0.282352} ) := 1.00 (0.001) 4
```

If any of the files to be loaded have already been loaded, they will each be unloaded before being re-loaded. See the command `unload` (Section 4.4 on page 39) to manually unload the *knowledge* from a given set of files.

reload

`/reload(`*string+*`)`

The `reload` command allows *knowledge* to be re-loaded from (properly formatted) text files. All terms in the `predicate` are expected to be *strings*.

```
?- /load("./etc/samples/manual.fizz")
load : loading ./etc/samples/manual.fizz ...
load : loaded  ./etc/samples/manual.fizz in 0.018s
?- /reload("./etc/samples/manual.fizz")
reload : unloading ./etc/samples/manual.fizz ...
reload : unloaded ./etc/samples/manual.fizz in 0.003s
reload : loading ./etc/samples/manual.fizz ...
reload : loaded  ./etc/samples/manual.fizz in 0.018s
```

poke

`/poke(`*symbol*`|`*string*`|`*guid,symbol*`,` *term*`)`

The `poke` command allows the *properties* of an *elemental* object to be written. For example, in the case of the `rand` *elemental* as defined in Section 2.5 on page 8, we can change the value of its `min` *properties* as follows:

```
?- /poke(rand,min,1545)
?- /peek(rand,min)
peek : min = 1545
```

In this example, as in the one for the `/peek` command, we have used the label of the *elemental* to identify it. If there are more than one *elemental* responding to the same label, they will all receive and process the `poke`. In such situation, we should have use the GUID of the *elemental* to only target a single one.

save

`/save(`*string*`,` *symbol**`)`

The `save` command allows *knowledge* to be saved to a (properly formatted) text file, allowing it to be re-loaded at a later time. The command supports saving all *knowledges* or a selection based on their *labels*. To save all existing *knowledges* currently in the *runtime* environment, you only need to provide the name of the text file to be created:

```
?- /save("all.fizz")
save: completed in 0.141s.
```

If we wanted to save only the `weather` *knowledges*, we would do:

```
?- /save("weather.fizz",weather)
save: completed in 0.04s.
```

All terms except the first one are expected to be *symbols*.

scan

`/scan`

The `scan` command will keep printing statistics on the *runtime environment* until none of the statistics changes in the *substrate*:

```
scan : e:11 k:7 s:2 p:7 u:3.49 t:11 q:3945 r:4384 z:0
scan : e:11 k:7 s:2 p:7 u:3.73 t:1 q:4471 r:5069 z:0 (qps:2191.7 rps:2854.2)
scan : e:11 k:7 s:2 p:7 u:3.98 t:4 q:4995 r:5793 z:0 (qps:2071.1 rps:2861.7)
scan : e:11 k:7 s:2 p:7 u:4.23 t:1 q:5503 r:6498 z:0 (qps:2056.7 rps:2854.3)
scan : e:11 k:7 s:2 p:7 u:4.48 t:2 q:6138 r:7401 z:0 (qps:2529.9 rps:3597.6)
scan : e:11 k:7 s:2 p:7 u:5.00 t:3 q:6843 r:8541 z:0 (qps:0.0 rps:3666.7)
scan : e:11 k:7 s:2 p:7 u:5.25 t:1 q:7789 r:9452 z:0 (qps:3814.5 rps:3673.4)
scan : e:11 k:7 s:2 p:7 u:5.50 t:4 q:8790 r:10426 z:0 (qps:3956.5 rps:3849.8)
```

The breakdown of the statistic is identical to the `stats` command with the addition of `qps` and `rps` which are respectively *queries per seconds* and *replies per seconds*.

spy

`/spy(append,`*symbol+*`)`
`/spy(remove,`*symbol+*`)`

Instructs the *runtime* to start or stop printing any events (queries, replies, ...) related to any of the *knowledge* labels provided as arguments. *Spying* is a handy way to see what is happening within the *runtime* and can be extremly useful to debug. In the following example, we *spy* on the `gameboy.color` *knowledge* then submit a query:

```
?- /spy(append,gameboy.color)
spy : observing gameboy.color
?- @gameboy.color({r = :r?[gt(0.1),lt(0.4)], g = :g, b = :b})
spy : [1589955735.839] Q @gameboy.color(:color) (14.999948)
spy : [1589955735.839] R gameboy.color({r = 0.509803, g = 0.784313, b = 0.294117}) (14.999855)
-> ( {r = 0.509803, g = 0.784313, b = 0.294117} ) := 1.00 (0.000) 1
spy : [1589955735.839] R gameboy.color({r = 0.325490, g = 0.670588, b = 0.392156}) (14.999855)
spy : [1589955735.839] R gameboy.color({r = 0.164705, g = 0.549019, b = 0.349019}) (14.999855)
-> ( {r = 0.325490, g = 0.670588, b = 0.392156} ) := 1.00 (0.000) 2
spy : [1589955735.839] R gameboy.color({r = 0, g = 0.294117, b = 0.282352}) (14.999855)
-> ( {r = 0.164705, g = 0.549019, b = 0.349019} ) := 1.00 (0.001) 3
-> ( {r = 0, g = 0.294117, b = 0.282352} ) := 1.00 (0.001) 4
```

Outputs from *spying* will always be prefixed with `spy`, followed by a timestamp after the colon. The following character indicates the type of the observed event:

`Q`	a query.
`R`	a reply.
`S`	a statement.
`T`	a query is being scrapped.

stats

`/stats`

Print to the *console* some basic statistic about what is happening in the runtime:

```
?- /stats
stats : e:2 k:1 s:0 p:0 u:1.29 t:1 q:0 r:0 z:0
```

The breakdown of the statistic is the following:

`e`	current number of *elemental* objects in the *substrate*.
`k`	total number of *knowledges* on the *substrate*.
`s`	total number of *statements* on the *substrate*.
`p`	total number of *prototypes* on the *substrate*.
`u`	up time (in seconds) of the *runtime*.
`t`	elapsed time (in miliseconds) it took for the statistics to be collected.
`q`	total number of *queries* posted on the *substrate*.
`r`	total number of *replies* (in *statements*) posted on the *substrate*.
`z`	total number of *statement* posted (without *query*) on the *substrate*.

tells

`/tells(`*symbol*`|`*string*`|`*guid*`,`*functor*`|`*symbol*`)`

Sends a *message* (in the form of a *functor* or a *symbol*) to an *elemental* object identified by its label, alias or GUID, the first argument. Not all *elemental* object can handle *message*. If the object is identified by its label, all objects with the same label will receive the message.

```
?- /tells(some.obj,do(this,45))
```

trace

`/trace(`*symbol*`,`*string?*`)`

The `trace` command supports controlling the builin tracing facility, which can be useful when debugging. The first *term*, a *symbol* specifies the tracing command to be executed:

`on`	turn the tracing ON.
`off`	turn the tracing OFF.
`print`	print to the console all recorded inference traces.
`clear`	clear all previously recorded inference traces.
`save`	save all previously recorded inference traces to a text file whoes path is provided as the second *term*.

Here's an example:

```
?- /trace(on)
trace - started
?- #surely_raining(:x)
-> ( paris ) := 0.80 (0.001) 1
-> ( mawsynram ) := 1.00 (0.001) 2
?- /trace(print)
Q: #surely_raining(:x)
    Q: @weather(:x, rain) = <0.700000|1>
    R: weather(paris, rain) {stamp = 1507093154.766867} := 0.80
    R: weather(mawsynram, rain) {stamp = 1507093176.743262} := 1.00
R: surely_raining(paris) := 0.80
R: surely_raining(mawsynram) := 1.00
Q: @weather(paris, sunny)
Q: @weather(mawsynram, sunny)
```

As shown above, the trace output render hierarchy by using tabulations. Please note that the tracing doesn't record primitive calls nor `self` predicates.

unload

`/unload(`*string+*`)`

The `unload` command allows *knowledge* loaded from a file to be unloaded. All terms in the `predicate` are expected to be *strings*.

```
?- /load("./samples/manual.fizz")
load : loading ./samples/manual.fizz ...
load : loaded  ./samples/manual.fizz in 0.011s
?- @gameboy.color(:color)
-> ( {r = 0.509803, g = 0.784313, b = 0.294117} ) := 1.00 (0.001) 1
-> ( {r = 0.325490, g = 0.670588, b = 0.392156} ) := 1.00 (0.001) 2
-> ( {r = 0.164705, g = 0.549019, b = 0.349019} ) := 1.00 (0.001) 3
-> ( {r = 0, g = 0.294117, b = 0.282352} ) := 1.00 (0.001) 4
?- /unload("./samples/manual.fizz")
unload : unloading ./samples/manual.fizz ...
unload : unloaded ./samples/manual.fizz in 0.000s
```

use

`/use(`*string+*`)`

The `use` command allows for one or more module(s) (shared library) to be loaded. All terms in the `predicate` are expected to be *strings*. Once loaded, the *module* contents will be available (e.g. *elemental classes*, *primitives*). A loaded *module* cannot be unloaded.

```
?- /use("modLGR")
use : loading ./mod/lnx/x64/modLGR.so ...
use : loaded  ./mod/lnx/x64/modLGR.so in 0.001s
?- /use("./modLGR.so")
use : sorry, ./modLGR.so doesn't exists
```

When no extension is given, the *command* assumes the *module* to be loaded is located in the *fizz* modules folder that correspond to the architecture used by the host computer.

wipe

`/wipe`

The `wipe` command will cause the *runtime* enviroment to be cleared of all existing *elementals* objects. The state of the *runtime* will be similar to the state at of the *runtime* when the executable is started.

peek

`/peek(`*symbol*|*string*|*guid*,*symbol*`)`

The `peek` command allows the *properties* of an *elemental* object to be read. For example, if we have a `rand` *elemental* as defined in Section 2.5 on page 8, we can read the value of its `min` *properties* as follows:

```
?- /peek(rand,min)
peek : min = 1550
```

5 Primitives

This Section details the *primitives* provided by the *runtime*. For each one, expected (and optional) arguments are described and for most a use case examples is given. All *primitives* are grouped under related categories.

5.1 Arithmetic

This section contains all the *primitives* that deal with basic *arithmetic*.

add

`add(`*number*`|`*variable*`,`*number*`|`*variable*`,`*number*`|`*variable*`)`

This *primitive* will unify or bind the sum of its two first *terms* with the third. For example:

```
?- add(4,3,:x)
-> ( 7 ) := 1.00 (0.001) 1
```

If the third *term* is a *number* or a *variable* bound to a *number*, one of the first *terms* can be an unbound *variable*. In that case the *primitive* will find the right value to make the addition valid as seen in the example below:

```
?- add(4,:x,7)
-> ( 3 ) := 1.00 (0.000) 1
```

div

`div(`*number*`|`*variable*`,`*number*`|`*variable*`,`*number*`|`*variable*`)`

This *primitive* will unify or bind the division of the first *term* by the second with the third. For example:

```
?- div(10,3,:x)
-> ( 3.333333 ) := 1.00 (0.000) 1
```

If the third *term* is a *number* or a *variable* bound to a *number*, one of the first *terms* can be an unbound *variable*. In that case the *primitive* will find the right value to make the division valid as seen in the following example:

```
?- div(:x,3,3.3333333)
-> ( 10.000000 ) := 1.00 (0.000) 1
```

div.int

`div.int(`*number*`|`*variable*`,`*number*`|`*variable*`,`*number*`|`*variable*`)`

This *primitive* will unify or bind the integer division of the first *term* by the second with the third. For example:

```
?- div.int(37,6,:x)
-> ( 6 ) := 1.00 (0.001) 1
```

If the third *term* is a *number* or a *variable* bound to a *number*, one of the first *terms* can be an unbound *variable*. In that case the *primitive* will find any values that will make the division valid as seen in the following example:

```
?- div.int(:v,6,5)
-> ( 30 ) := 1.00 (0.001) 1
-> ( 31 ) := 1.00 (0.001) 2
-> ( 32 ) := 1.00 (0.001) 3
-> ( 33 ) := 1.00 (0.001) 4
-> ( 34 ) := 1.00 (0.002) 5
-> ( 35 ) := 1.00 (0.002) 6
```

inv

`inv(`*number*`|`*variable*`,`*number*`|`*variable*`)`

This *primitive* will unify or bind the inverse value of the first *term* with the second. For example:

```
?- inv(4,:x)
-> ( -4 ) := 1.00 (0.000) 1
?- inv(:x,4)
-> ( -4 ) := 1.00 (0.000) 1
```

max

`max(`*number+*`,`*number*`|`*variable*`)`
`max(`*list*`,`*number*`|`*variable*`)`

This *primitive* will unify its last *term* with the maximum value in all its other *terms*. If the *primitive* as only two *terms* and the first *term* is a list, the maximum value in the *list* will be unified with the second *term*. For example:

```
?- max(3,2,-2,5,:min)
-> ( 5 ) := 1.00 (0.000) 1
?- max([3,2,-2,5],:min)
-> ( 5 ) := 1.00 (0.000) 1
```

min

`min(`*number+*`,`*number*`|`*variable*`)`
`min(`*list*`,`*number*`|`*variable*`)`

This *primitive* will unify its last *term* with the minimum value in all its other *terms*. If the *primitive* as only two *terms* and the first *term* is a list, the minimum value in the *list* will be unified with the second *term*. For example:

```
?- min(3,2,-2,5,:min)
-> ( -2 ) := 1.00 (0.000) 1
?- min([3,2,-2,5],:min)
-> ( -2 ) := 1.00 (0.000) 1
```

mod

`mod(`*number*`,`*number*`,`*number*`|`*variable*`)`

This *primitive* will unify or bind results from performing an integer division between the first two *terms* with the third. For example:

```
?- mod(9,2,:v)
-> ( 1 ) := 1.00 (0.000) 1
?- mod(8,2,:v)
-> ( 0 ) := 1.00 (0.000) 1
```

The *primitive* doesn't support the first or second term as unbound variables.

mul

`mul(`*number*`|`*variable*`,`*number*`|`*variable*`,`*number*`|`*variable*`)`

This *primitive* will unify or bind the multiplication of the first two *terms* with the third. For example:

```
?- mul(10,3,:x)
-> ( 30 ) := 1.00 (0.000) 1
```

If the third *term* is a *number* or a *variable* bound to a *number*, one of the first *terms* can be an unbound *variable*. In that case the *primitive* will find the right value to make the multiplication valid as seen in the following example:

```
?- mul(10,:x,4)
-> ( 0.400000 ) := 1.00 (0.000) 1
```

sim

`sim(`*number*`,`*number*`,`*number*`|`*variable*`)`

This *primitive* will unify its third *term* with a value representing the similarity between the first two *terms*. For example:

```
?- sim(3.21,3.33,:s)
-> ( 0.981651 ) := 1.00 (0.000) 1
?- sim(3.21,10,:s)
-> ( 0.485995 ) := 1.00 (0.000) 1
?- sim(3.21,-100,:s)
-> ( 0 ) := 1.00 (0.000) 1
?- sim(3.21,2.211,:s)
-> ( 0.815717 ) := 1.00 (0.000) 1
```

sub

`sub(`*number*`|`*variable*`,`*number*`|`*variable*`,`*number*`|`*variable*`)`

This *primitive* will unify or bind the second *term* subtracted from the first one with the third. For example:

```
?- sub(10,4,:x)
-> ( 6 ) := 1.00 (0.000) 1
```

If the third *term* is a *number* or a *variable* bound to a *number*, one of the first *terms* can be an unbound *variable*. In that case the *primitive* will find the right value to make the subtraction valid as seen in the following example:

```
?- sub(10,:x,4)
-> ( 6 ) := 1.00 (0.000) 1
```

sum

`sum(`*number+*`,`*number*`|`*variable*`)`

This *primitive* will unify or bind the sum of all *terms* with the last *term*. For example:

```
?- sum(3,3,6,7,:sum)
-> ( 19 ) := 1.00 (0.000) 1
?- sum(3,3,6,7,19)
-> (  ) := 1.00 (0.000) 1
```

Countrary to the *primitive* `add`, this *primitive* does not support having any *term* unbound but the last one.

5.2 Basic

Under this grouping are all the *primitives* that provide very basic - and in most cases essentials - capabilities to the *runtime*.

any

`any(`*term+*`,`*variable*`)`

This *primitive* will unify its last *term* with the first *term* that isn't a unbounded *variable*. For example:

```
?- set(:V,4), any(:V,2,:d)
-> ( 4 ) := 1.00 (0.000) 1
?- any(:V,2,:d)
-> ( 2 ) := 1.00 (0.000) 1
```

assert

`assert(`*functor*`,`*number*`,`*frame?*`)`
`assert(`*symbol*`,`*list*`,`*number*`,`*frame?*`)`

The `assert` *primitive* allows for a *statement* to be added to an existing *knowledge*. If no *elemental* object capable of handling it exists, the *runtime* will instantiate one. The following example shows how a new *statement* is added at *runtime* to the `weather` *knowledge*:

```
?- @weather(seattle,:s)
-> ( sunny ) := 0.20 (0.001) 1
?- assert(weather(seattle,rain),0.6)
-> (  ) := 1.00 (0.001) 1
?- @weather(seattle,:s)
-> ( sunny ) := 0.20 (0.001) 1
-> ( rain ) := 0.60 (0.001) 2
```

The optional third *term* to the *primitive* is a *frame* which (as we have seen in section 2.2 on page 3) provides the properties of the *statement*. Here's how we could timestamp each *statement* when asserting them:

```
?- assert(weather(paris,rain),0.8,{stamp = %now})
-> (  ) := 1.00 (0.000) 1
?- assert(weather(seattle,sunny),0.2,{stamp = %now})
-> (  ) := 1.00 (0.000) 1
?- assert(weather(london,fog),0.9,{stamp = %now})
-> (  ) := 1.00 (0.000) 1
?- assert(weather(mawsynram,rain),1,{stamp = %now})
-> (  ) := 1.00 (0.000) 1
?- assert(weather(honolulu,snow),0,{stamp = %now})
-> (  ) := 1.00 (0.000) 1
```

When a statement is *asserted*, it will be broadcasted in the *substrate*. See *primitive* `repeal` for the inverse function.

break

`break(`*boolean*`)`

The *primitive* `break` will prematurely end an ongoing inference when its *term* unify to the boolean value *true*. The call will always evaluate to a *truth value* of `1.0`.

```
?- console.puts(a), break(1), console.puts(b)
a
-> (  ) := 1.00 (0.000) 1
?- console.puts(a), break(0), console.puts(b)
a
b
-> (  ) := 1.00 (0.000) 1
```

See the sample `leibniz.fizz` for an example of its use.

break.not

`break.not(`*boolean*`)`

The *primitive* `break` will prematurely end an ongoing inference when its *term* unify to the boolean value *false*. The call will always evaluate to a *truth value* of `1.0`.

```
?- console.puts(a), break.not(0), console.puts(b)
a
-> (  ) := 1.00 (0.000) 1
?- console.puts(a), break.not(1), console.puts(b)
a
b
-> (  ) := 1.00 (0.000) 1
```

bundle

`bundle(`*functor*, *number*, *frame*, *number?*`)`
`bundle(`*symbol*, *list*, *number*, *frame*, *number?*`)`

Like the `assert` *primitive*, `bundle` allows for a *statement* to be added to an existing *knowledge*. It however provides a way for the *statements* provided during consecutive (or concurrent) calls to be grouped into a single *knowledge*. Once a specified number of *statements* have been reached, or if the time elapsed since the last addition of a *statement* reaches a timeout value, the *knowledge* will be asserted into the *substrate*. In the following example, we define a *procedural knowledge* which when triggered (by any `line.f` *statement*) will assert a `frag` *statement* bundled within *knowledges* of 1024 *statements* in size:

```
import.frag {

    () :- @line.f(:i,:s), bundle(frag(:i,:s),1,{},1024), hush;

}
```

If the last *term* isn't given, the default value specified in the *runtime* settings (`bundle.len`) will be used.

cache

`cache(peek,`*atom*`,`*term*`|`*variable*`,`*term?*`)`
`cache(poke,`*atom*`,`*term*`)`
`cache(push,`*atom*`,`*term*`)`
`cache(pull,`*atom*`,`*term*`|`*variable*`)`
`cache(drop,`*atom*`)`

This *primitive* provides a synchronized access to a global storage area (host only) where *terms* can be stored and retreived based on a key (any *atom* can be a key). When the first *term* unifies to the *symbol* `peek`, the value associated with the key will be unified against the third *term*. If the key doesn't exists and a fourth *term* was provided to the *primitive*, that value will be unified against the third *term* instead. When the first *term* unifies to the *symbol* `poke`, the second *term* will either set or replace the value stored for the provided key. If the first *term* is `drop`, any value stored for the key will be removed from the cache. For example:

```
?- cache(poke,hello,42.5)
-> (  ) := 1.00 (0.000) 1
?- cache(peek,hello,:v)
-> ( 42.500000 ) := 1.00 (0.000) 1
?- cache(peek,hell0,:v,5)
-> ( 5 ) := 1.00 (0.000) 1
```

If the first *term* is `push`, the key will be treated as referencing a queue, and the third *term* will be pushed onto the queue. When the first *term* is `pull`, the next *term* on the queue will be removed from it and unified to the third *term* of the *primitive*. If the queue is empty or if the key doesn't exist, the *primitive* call will evaluate to a *truth value* of `0`. For example:

```
?- rng.span(<0|1>,0.1,:I), cache(push,q,:I)
-> (  ) := 1.00 (0.001) 1
?- cache(pull,q,:v)
-> ( 0 ) := 1.00 (0.000) 1
?- cache(pull,q,:v)
-> ( 0.100000 ) := 1.00 (0.000) 1
?- cache(pull,q,:v)
-> ( 0.200000 ) := 1.00 (0.000) 1
?- cache(pull,q,:v)
-> ( 0.300000 ) := 1.00 (0.000) 1
```

If the first *term* is a *functor* instead of just a *symbol*, the first *atom* in its *terms* will be used as a key representing a different caching context than the default one. For example:

```
?- cache(poke(a),v,1)
-> (  ) := 1.00 (0.000) 1
?- cache(poke(b),v,1)
-> (  ) := 1.00 (0.000) 1
?- cache(poke(c),v,2)
-> (  ) := 1.00 (0.000) 1
?- cache(poke(0),v,3)
-> (  ) := 1.00 (0.000) 1
?- cache(peek(a),v,:v)
-> ( 1 ) := 1.00 (0.000) 1
?- cache(peek(b),v,:v)
-> ( 1 ) := 1.00 (0.000) 1
?- cache(peek(c),v,:v)
-> ( 2 ) := 1.00 (0.000) 1
?- cache(peek(0),v,:v)
-> ( 3 ) := 1.00 (0.000) 1
```

cease

`cease(`*symbol+*`)`
`cease(`*list*`)`

This *primitive* can be used to remove one or more *elementals* from the *runtime* using their labels. Here's an example:

```
?- spawn(tick,{class = FZZCTicker, tick = 0.5, tick.on.attach  = yes})
-> (  ) := 1.00 (0.001) 1
?- /spy(append,tick)
spy : observing tick
spy : [1589697893.355] S tick(9, 1589697893.355115) (15.000000)
spy : [1589697893.856] S tick(10, 1589697893.855659) (15.000000)
spy : [1589697894.355] S tick(11, 1589697894.355373) (15.000000)
spy : [1589697894.855] S tick(12, 1589697894.854866) (15.000000)
spy : [1589697895.356] S tick(13, 1589697895.355578) (15.000000)
spy : [1589697895.855] S tick(14, 1589697895.855147) (15.000000)
spy : [1589697896.355] S tick(15, 1589697896.354837) (15.000000)
?- cease(tick)
spy : [1589697896.856] S tick(16, 1589697896.855403) (15.000000)
spy : [1589697897.355] S tick(17, 1589697897.355099) (15.000000)
spy : [1589697897.856] S tick(18, 1589697897.855685) (15.000000)
spy : [1589697898.355] S tick(19, 1589697898.355348) (15.000000)
spy : [1589697898.855] S tick(20, 1589697898.854917) (15.000000)
spy : [1589697899.356] S tick(21, 1589697899.355585) (15.000000)
spy : [1589697899.855] S tick(22, 1589697899.855141) (15.000000)
spy : [1589697900.355] S tick(23, 1589697900.354838) (15.000000)
spy : [1589697900.855] S tick(24, 1589697900.855344) (15.000000)
spy : [1589697901.355] S tick(25, 1589697901.355057) (15.000000)
spy : [1589697901.856] S tick(26, 1589697901.855657) (15.000000)
spy : [1589697902.355] S tick(27, 1589697902.355204) (15.000000)
spy : [1589697902.855] S tick(28, 1589697902.854851) (15.000000)
spy : [1589697903.356] S tick(29, 1589697903.355564) (15.000000)
spy : [1589697903.855] S tick(30, 1589697903.855035) (15.000000)
spy : [1589697904.356] S tick(31, 1589697904.355737) (15.000000)
-> (  ) := 1.00 (0.000) 1
```

change

`change(`[*functor*,*number?*,*frame?*]`,`[*functor*,*number?*,*frame?*]`)`
`change(`[*symbol*,*list*,*number?*,*frame?*]`,`[*symbol*,*list*,*number?*,*frame?*]`)`

The `change` *primitive* combines a `repeal` followed by an `assert`. In the following example, we use it to replace an earlier version of the *statement* with one with the current time:

```
?- change([city.weather.latest(:id,_)],[city.weather.latest(:id,%now)])
```

Both terms are expected to be *lists*, describing the *statement* to be repealed and the *statement* to be asserted (as per the primitives `repeal` and `assert`).

console.exec

`console.exec(`*atom*|*functor*`)`

This *primitive* will trigger the background execution of a console's *command*. It can be used, for instance by an elemental to trigger the frequent saving of all (or selected) knowledge during the execution. Here's an example:

```
?- console.exec(bye)
-> (  ) := 1.00 (0.000) 1
bye!
```

console.gets

`console.gets(`*variable*`)`
`console.gets(`*term*,*variable*`)`

This *primitive* will read a line from the console. Since the user will be prompted to enter a string as a synchronous operation, calling this *primitive* will only work when offloaded. If two *terms* are given, the first one will be assumed to be something to be printed before (e.g. a prompt). For example:

```
?- &console.gets(:x)
>- hello world!
-> ( "hello world!" ) := 1.00 (5.105) 1
?- &console.gets("Tell me your name:",:s)
Tell me your name:
>- Roger
-> ( "Roger" ) := 1.00 (5.405) 1
```

The *primitive* will also publish a *statement* when the input is validated by the user.

console.puts

`console.puts(`*term+*`)`

This *primitive* will output the concatenation of the terms in the console. For example:

```
?- console.puts(Hello,"", world","!")
Hello, world!
```

console.quit

`console.quit()`

This *primitive* will request the console application to quit.

cpy

cpy(*term*,*variable*,*symbol?*)

This *primitive* will unify its last *term* with a copy of the first *term* where every unbound *variables* found in the first *term* will be replaced by a different instance of the *variable*. This is make clear in the following example: the difference between `set` and `cpy` can be observed as the *term* bound to the *variable* `l` doesn't contains the values bound to the *variables* `A` and `B` after `cpy` was called, contrasting with the effect`set` had:

```
?- set(:L,[:A,:B]), set(:L,:l), set(:A,1), set(:B,2)
-> ( [1, 2] ) := 1.00 (0.000) 1
?- set(:L,[:A,:B]), cpy(:L,:l), set(:A,1), set(:B,2)
-> ( [:A, :B] ) := 1.00 (0.000) 1
```

If a third *term* is provided and is the *symbol* `local`. The copied *variables* will be local to the calling context.

cut.if

cut.if(*number*)

This *primitive* will have the same effect as using the caret (^) after a *predicate*, but only if its only *term* unifies with the value `1`.

cut.if.not

cut.if.not(*number*)

This *primitive* will have the same effect as using the caret (^) after a *predicate*, but only if its only *term* unifies with the value `0`.

declare

declare(*list+*)
declare(*functor*,*number?*,*frame?*)
declare(*symbol*,*list*,*number?*,*frame?*)

This *primitive* will broadcast statements into the *runtime environment* built from its *terms*. A *functor* (or a *symbol* plus a *list*) followed by an optional *truth value* and an optional *frame* is requiered for the *primitive* to create a *statement*. Multiple statements can be broadcasted if they are enclosed in lists. For example:

```
?- /spy(append,blah)
spy : observing blah
?- declare(blah(23,hello))
spy : S blah(23, hello) := 1.00
-> (  ) := 1.00 (0.001) 1
?- declare([blah(23,hello)],[blah(25,bye)])
spy : S blah(23, hello) := 1.00
spy : S blah(25, bye) := 1.00
-> (  ) := 1.00 (0.002) 1
?- declare([blah(23,hello),0.8],[blah(25,bye),0.5])
spy : S blah(23, hello) := 0.80
spy : S blah(25, bye) := 0.50
```

```
-> (  ) := 1.00 (0.002) 1
?- declare([blah(23,hello),0.8],[blah(25,bye),0.5,{stamp = %now}])
spy : S blah(23, hello) := 0.80
spy : S blah(25, bye) := 0.50 {stamp = 1507446180.615446}
-> (  ) := 1.00 (0.002) 1
?- declare([[blah(23,hello),0.8],[blah(25,bye),0.5,{stamp = %now}]])
spy : S blah(23, hello) := 0.80
spy : S blah(25, bye) := 0.50 {stamp = 1507446211.905603}
-> (  ) := 1.00 (0.000) 1
```

If multiple *statements* have the same label, they will be grouped according to the *runtime environment*'s *sspr* value and broadcasted together.

define

`define(`*symbol*, *list*, *list*, *list*`)`

The `define` *primitive* allows for a *prototype* to be added to the *knowledge* contained on the *substrate*. If no *elemental* object capable of handling it exists, the *runtime* will instantiate one. The following example defines two *prototypes* which together print the content of a list given as input:

```
?- define(lst.print,[[]],[cut],[[[primitive],true()]])
-> (  ) := 1.00 (0.000) 1
?- define(lst.print,[[:h|:t]],[],[[[primitive],console.puts(:h)],[[],[lst.print,[:t]]]])
-> ( :h , :t ) := 1.00 (0.000) 1
?- #lst.print([a,b,c])
a
b
c
-> (  ) := 1.00 (0.002) 1
```

This would have had the same result as defining the `lst.print` *knowledge* as:

```
lst.print {

    ([]) ^ :- true;
    ([:h| :t]) :- console.puts(:h), #lst.print(:t);

}
```

The first *term* is the label of the *prototype*, followed by a *list* containing the *entrypoint*. The third *term* is a list of options (for example the *symbol cut* to turns the *prototype* into *a cut* one). The last *term* is a list containing the definitons of all the *predicates* that makes up the *prototype*. Each of the *predicate* is it-self defined within a list. As shown in the above example, this *list* is expected to have two *elements*. The first one is a list of options (symbols such as `negate, primitive, cut, offload, trigger`. The list can also contain a *range term* and a *frame term*. The second *term* can either be a *functor* or a *list* containing the label of the *predicate* and a list of the *predicate*'s *terms*.

See the *primitive* `revoke` for the inverse effect in Section 5.2 on page 54.

drop

`drop(`*symbol*`)`

The `drop` *primitive* allows for a *property* of the calling *elemental* object to be removed. Please note that offloading the execution of the *primitive* will not work.

exec

`exec(`*symbol*, *list*`)`
`exec(`*functor*`)`

This *primitive* can be used to execute an arbitraty *primitive* specified by a *symbol* and a *list* of *terms*, or as a *functor*. Here's an example:

```
?- exec(add(2,3,:v))
-> ( 5 ) := 1.00 (0.001) 1
?- exec(add,[2,3,:v])
-> ( 5 ) := 1.00 (0.000) 1
```

false

`false`
`false(`*boolean*`|`*variable*`)`

Calling this *primitive* with no *term* will cause the on-going *inference* to fail by resolving to a *truth value* of `0`. When used with a single *term* it will either test of a value is *false* or bind a *variable* to the value `false`.

forget

`forget(`*symbol+*`)`

The `forget` *primitive* will cause all *elemental* objects with the label given in its *terms* to be removed from the *substrate*.

```
?- forget(product,product.g)
-> (  ) := 1.00 (0.000) 1
```

fuzz

`fuzz(`*number*`)`

The `fuzz` *primitive* will resolve with a *truth value* during *inference* the value passed as term:

```
?- fuzz(0.2)
-> (  ) := 0.20 (0.000) 1
```

hash

`hash(`*term*, *variable*`)`

The `hash` *primitive* will unify and/or substitute its second *term* with an hashcode computed from its first *term*.:

```
?- hash(["hello",world],:h)
-> ( 3087048980 ) := 1.00 (0.000) 1
```

hush

`hush`

The *primitive* `hush` will husher the ongoing inference. No *statement* will be published and no query will be answered. This is useful mainly in situations where a *prototype* is activated by a *trigger* predicate.

hush.if

`hush.if(`*number*`)`

Just like `hush`, this *primitive* will husher the ongoing inference, but only if its only *term* unifies with the value `1`.

hush.if.not

`hush.if.not(`*number*`)`

Just like `hush`, this *primitive* will husher the ongoing inference, but only if its only *term* unifies with the value `0`.

nab

`nab(`*term*`,`*variable*`)`

This *primitive* will unify its second *term* with an unescaped version of the first *term*. For example:

```
?- set(:B,5), nab(\[1,a,:B],:l)
-> ( [1, a, 5] ) := 1.00 (0.000) 1
```

now

`now(`*number*`|`*variable*`)`

This *primitive* will unify and/or substitute its sole *term* with the current host time (UTC, expressed in seconds since Unix epoch).

peek

`peek(`*symbol*`,`*variable*`|`*term*`)`

The `peek` *primitive* allows for a *property* of the calling *elemental* object to be read and unified and/or substitued with the second *term*. If the label provided as the first *term* is not a known *property*, the call will evaluate to a *truth value* of `0.0`. For example, the following *knowledge* will multiply a value by a factor read from its *properties*:

```
multiplier { factor = 2 } {

    (:v,:v2) :- peek(factor,:f), mul(:v,:f,:v2);

}
```

Using the *console command* `/poke` we can modify the value of the *knowledge* property on the fly as shown here:

```
?- #multiplier(3,:v)
-> ( 6 ) := 1.00 (0.002) 1
?- /poke(multiplier,factor,3)
?- #multiplier(3,:v)
-> ( 9 ) := 1.00 (0.002) 1
```

Accessing *properties* during inferences can allow for an easier reuse of *knowledge*. Please note that this *primitive* will not work when offloaded.

poke

`poke(`*symbol*, *term*`)`

The `poke` *primitive* allows for a *property* of the calling *elemental* object to be written with the second *term* as value. If the label provided as the first *term* is not a known *property* or if it is a reserved label (like `class guid label alias`), the call will evaluate to a *truth value* of `0.0`. Changing the value of a *property* during inference supports allow for the *elemental* to save states. The following example uses two *properties* to cycle through a list of words to only return a different word at each inference:

```
wword {

    index = 0,
    words = [when, why, where, how]

} {

    // the prototype will reset the index to 0 if its value is the size of the words list
    (:w) :- peek(index,:i),
            peek(words,:l),
            lst.length(:l,:s),
            eq(:i,:s),
            poke(index,0),
            false;

    // the main prototype
    (:w) :- peek(index,:i),
            peek(words,:l),
            lst.item(:l,:i,:w),
            add(:i,1,:i2),
            poke(index,:i2);

}
```

Just like with the `peek` *primitive*, offloading the execution of the *primitive* will not work.

prune

`prune`

The *primitive* `prune` will instruct the *solver* to prune any other concurrent solving or possible backtracking steps.

pull

`pull(`*symbol*, *variable*`|`*term*`)`

The `pull` *primitive* allows for a *property* of the calling *elemental* object to be treated as a queue from which a value can be pulled, unified and/or substitued with the second *term*. If the label provided as the first *term* is not a known *property* or if the queue is empty, the call will evaluate to a *truth value* of `0.0`. For example, the following *knowledge* accumulate *numbers* in a queue and get the maximum value on demand:

```
accumulator { values = [] } {

    (dec,:v)    :-  ?pull(values,:h), any(:h,0,:v);
    (acc,:v)    :-  push(values,:v);
    (max,:m)    :-  lst.max($values,:m);

}
```

See the *primitive* `push` for the inverse effect.

```
?- #accumulator(acc,1)
-> (  ) := 1.00 (0.001) 1
?- #accumulator(acc,2)
-> (  ) := 1.00 (0.001) 1
?- #accumulator(acc,3)
-> (  ) := 1.00 (0.001) 1
?- #accumulator(acc,4)
-> (  ) := 1.00 (0.001) 1
?- #accumulator(max,:m)
-> ( 4 ) := 1.00 (0.001) 1
?- #accumulator(dec,:v)
-> ( 1 ) := 1.00 (0.001) 1
```

Please note that this *primitive* will not work when offloaded.

push

`push`(*symbol*, *term*)

The `push` *primitive* allows for a *property* of the calling *elemental* object to be treated as a queue from which a value can be pushed onto. If the property already exist and it isn't a *list*, the previous value will be stored first in the queue. Please note that this *primitive* will not work when offloaded.

repeal

`repeal`(*functor*, *number*)
`repeal`(*symbol*, *list*, *number*)

The `repeal` *primitive* allows for a *statement* to be removed from any existing *knowledge*. If the *functor* or the *terms list* contains unbound variables, any matching *statements* will be removed.

```
?- @weather(seattle,:s)
-> ( sunny ) := 0.20 (0.005) 1
-> ( rain ) := 0.60 (0.008) 2
?- repeal(weather,[seattle,rain],0.6)
-> (  ) := 1.00 (0.000) 1
?- @weather(seattle,:s)
-> ( sunny ) := 0.20 (0.005) 1
```

Note that the *elemental* object that was storing the *statement* will not be detached from the *substrate* even if it doesn't hold any more *knowledge*.

revoke

`revoke(`*symbol*,*list*,*list*,*list*`)`

The `revoke` *primitive* allows for a *prototype* to be removed from the *knowledge* contained on the *substrate*. It is the reverse action of the *primitive* `define` (see Section 5.2 on page 49). Using the example from that *primitive* we can remove both *prototypes* as follow:

```
?- revoke(lst.print,[[]],[cut],[[[primitive],true()]])
-> (  ) := 1.00 (0.000) 1
?- revoke(lst.print,[[:h|:t]],[],[[[primitive],console.puts(:h)],[[],[lst.print,[:t]]]])
-> ( :h , :t ) := 1.00 (0.000) 1
?- #lst.print([a,b,c])
```

Note that the *elemental* object that was storing the *prototype* will not be detached from the *substrate* even if it doesn't hold any more *knowledge*.

sleep

`sleep(`*number*`)`

This *primitive* will cause the calling *elemental* to suspend its execution for a given number of miliseconds.

set

`set(`*term*,*term*`)`

The `set` *primitive* primary use is to assign a value to a *variable*, but it can also be used to unify *terms* or *variables*. When used in the former case, the order in the *terms* doesn't matter as shown in the example below:

```
?- set(:x,4)
-> ( 4 ) := 1.00 (0.000) 1
?- set(4,:x)
-> ( 4 ) := 1.00 (0.000) 1
```

set.if

`set.if(`*term*,*term*,*boolean*`)`

The `set.if` *primitive* functions as the *primitive* `set` but only if its third *term* is a *number* which boolean value is *true*. If it's *false*, it will evaluate to a *truth value* of `1.0` and the *variable* will not be bound. For example:

```
?- set.if(5,:v,1)
-> ( 5 ) := 1.00 (0.000) 1
?- set.if(5,:v,0)
-> ( :v ) := 1.00 (0.000) 1
?- set.if(5,:v,0), set(6,:v)
-> ( 6 ) := 1.00 (0.000) 1
```

set.if.not

`set.if.not`(*term*, *term*, *boolean*)

The `set.if` *primitive* functions as the *primitive* `set` but only if its third *term* is a *number* which boolean value is *false*. If it's *true*, it will evaluate to a *truth value* of `1.0` and the *variable* will not be bound. For example:

```
?- set.if.not(5,:v,0)
-> ( 5 ) := 1.00 (0.000) 1
?- set.if.not(5,:v,1)
-> ( :v ) := 1.00 (0.000) 1
```

shoot

`shoot`(*symbol*,*list*)
`shoot`(*list*,*list*)

This *primitive* will cause a predicate inquery just like a *predicate* would, but will not wait for reply. If the first *term* is a *list*, it is supposed to contains the labels of the predicate to be created. The second *term* is expected the be the list of *terms*, to pass as the *predicate*'s *terms*. For example, we have the following *knowledge*:

```
test {

    (1,:str)    :-  console.puts(:str);
    (2,:str)    :-  str.toupper(:str,:s), console.puts(:s);

}
```

We would use the *primitive* as follow:

```
?- shoot(test,[2,"hello"])
-> (  ) := 1.00 (0.000) 1
HELLO
?- shoot(test,[1,"hello"])
-> (  ) := 1.00 (0.000) 1
hello
```

spawn

`spawn`(*symbol*, *frame*)
`spawn`(*symbol*, *frame*, *symbol*)

This *primitive* can be used to spawn a new *elemental* into the *runtime*. The first *term* of the *primitive* is the label of the *elemental* and the second *term* is its properties. Here's an example:

```
?- /spy(append,tick)
spy : observing tick
?- spawn(tick,{class = FZZCTicker, tick = 0.5, tick.on.attach  = yes})
-> (  ) := 1.00 (0.001) 1
spy : [1589697647.775] S tick(1, 1589697647.775018) (15.000000)
spy : [1589697648.175] S tick(2, 1589697648.174596) (15.000000)
spy : [1589697648.675] S tick(3, 1589697648.675118) (15.000000)
```

```
spy : [1589697649.175] S tick(4, 1589697649.174807) (15.000000)
spy : [1589697649.676] S tick(5, 1589697649.675423) (15.000000)
spy : [1589697650.175] S tick(6, 1589697650.175078) (15.000000)
spy : [1589697650.676] S tick(7, 1589697650.675677) (15.000000)
spy : [1589697651.175] S tick(8, 1589697651.175286) (15.000000)
```

If a third *term* is provided, it will be considered the be the *label* of an existing *elemental* whoes *knowledge* must be copied into the new *elemental*.

then

`then`(*number*|*variable*,*number*|*variable*,*number*|*variable*,*number*|*variable+*)

This *primitive* will unify and/or substitute its last *term* with the date/time (UTC, expressed in seconds since Unix epoch) build from the other *terms*. The first time is expected to be the calendar *year*, followed by the *month* and the *day*. Following optional *terms* are, in order: *hours*, *minutes*, *seconds* and *miliseconds*. For example:

```
?- then(:y,:m,:d,%now)
-> ( 2017 , 12 , 14 ) := 1.00 (0.001) 1
?- then(:y,:m,:d,:h,:min,%now)
-> ( 2017 , 12 , 14 , 20 , 12 ) := 1.00 (0.001) 1
?- then(:y,:m,:d,:h,:min,:s,:ms,%now)
-> ( 2017 , 12 , 14 , 20 , 12 , 21 , 713 ) := 1.00 (0.001) 1
?- then(2018,1,1,:new_year)
-> ( 1514764800 ) := 1.00 (0.001) 1
```

tme.str

`tme.str`(*number*|*variable*,*string*|*variable*)

This *primitive* will unify and/or substitute its *terms* in between a date/time (UTC, expressed in seconds since Unix epoch) and a string representation of that date. The first *term* is expected to be either a *number* or a *variable* and the second either a *string* or a *variable*. For example:

```
?- tme.str(%now,:s)
-> ( "Thu, 22 Feb 2018 06:58:23 GMT" ) := 1.00 (0.000) 1
?- tme.str(:t,"Thu, 22 Feb 2018 06:58:23 GMT")
-> ( 1519282703 ) := 1.00 (0.000) 1
```

true

`true`
`true`(*boolean*|*variable*)

Calling this *primitive* will cause the *inference* to continue. This is sort of a *no-op* with limited use, except to turn a *statement* into a *prototype*. When it is used with a single *term* it will either test if a value is *true* or bind a *variable* to the value `true`.

uny

`uny`(*term*,*term*)

This *primitive* will unify its first *term* with its second *term* without actually performing the unification. If both *terms* unifies, then the call will resolve with a *truth value* of `1`. Otherwise the *truth value* will be `0`. For example:

```
?- uny(a,a)
-> (  ) := 1.00 (0.000) 1
?- uny(a,b)
-> (  ) := 0.00 (0.000) 1
?- uny(a,:l)
-> ( :l ) := 1.00 (0.000) 1
```

whisper

`whisper(`*functor*, *number*, *frame?*`)`
`whisper(`*symbol*, *list*, *number*, *frame?*`)`

The `whisper` *primitive* allows for a *statement* to be added to an existing *knowledge*. If no *elemental* object capable of handling it exists, the *runtime* will instantiate one. The following example shows how a new *statement* is added at *runtime* to the `weather` *knowledge*:

```
?- @weather(seattle,:s)
-> ( sunny ) := 0.20 (0.001) 1
?- whisper(weather(seattle,rain),0.6)
-> (  ) := 1.00 (0.001) 1
?- @weather(seattle,:s)
-> ( sunny ) := 0.20 (0.001) 1
-> ( rain ) := 0.60 (0.001) 2
```

Unlike with `assert`, when a statement is *whispered*, it will not be broadcasted in the *substrate*. See *primitive* `repeal` for the inverse function.

5.3 Comparaisons

All *primitives* related to comparing two *terms* are grouped in this category.

aeq

`aeq(`*number*, *number*, *number*`)`

This *primitive* will evaluate to a *truth value* of `1.0` if its two first *terms* are almost equal *numbers*, and `0.0` if they do not. The third *term* is the maximum allowed difference between the two *numbers* to be estimated to be the same. For example:

```
?- aeq(4.5,4.51,0.01)
-> (  ) := 1.00 (0.001) 1
?- aeq(4.5,4.52,0.01)
-> (  ) := 0.00 (0.000) 1
```

are.different

`are.different(`*term*, *term*`)`

This *primitive* will evaluate to a *truth value* of `1.0` if its two *terms* do not unify, and `0.0` if they do.

are.same

`are.same(`*term*, *term*`)`

This *primitive* will evaluate to a *truth value* of `1.0` if its two *terms* do unify, and `0.0` if they don't.

cmp

`cmp(`*term*`,` *term*,*variable*`|` *term*`)`

This *primitive* will unify or bind the comparison (lesser, greater or equal) between the first two *terms* with the third. For example:

```
?- cmp(4,3,:c)
-> ( 1 ) := 1.00 (0.000) 1
?- cmp(2,3,:c)
-> ( -1 ) := 1.00 (0.000) 1
?- cmp(hello,hello,:c)
-> ( 0 ) := 1.00 (0.000) 1
```

eq

`eq(`*term*`,` *term*`)`
`eq(`*term*`,` *term*`,` *boolean*`|` *variable*`)`

This *primitive* will evaluate to a *truth value* of `1.0` if its two *terms* do unify, and `0.0` if they don't. It is a *short hand* to the `are.same` *primitive*. When used with three *terms*, the *primitive* will always evaluate to a *truth value* of `1.0` if its third *term* unify with the boolean value coming from the succes of the unification of the 2 first *terms*. For example:

```
?- eq(3,5,:e)
-> ( false ) := 1.00 (0.000) 1
?- eq(3,3,:e)
-> ( true ) := 1.00 (0.000) 1
```

gt

`gt(`*term*`,` *term*`)`

This *primitive* will evaluate to a *truth value* of `1.0` if the first *term* is a *number* and has a value greater than the second *term*, also a *number*. In all other cases, the *primitive* will evaluate to `0.0`.

gte

`gte(`*term*`,` *term*`)`

This *primitive* will evaluate to a *truth value* of `1.0` if the first *term* is a *number* and has a value greater or equal to the second *term*, also a *number*. In all other cases, the *primitive* will evaluate to `0.0`.

lt

`lt(`*term*`,` *term*`)`

This *primitive* will evaluate to a *truth value* of `1.0` if the first *term* is a *number* and has a value lesser than the second *term*, also a *number*. In all other cases, the *primitive* will evaluate to `0.0`.

lte

`lte(`*term*`,` *term*`)`

This *primitive* will evaluate to a *truth value* of `1.0` if the first *term* is a *number* and has a value lesser or equal to the second *term*, also a *number*. In all other cases, the *primitive* will evaluate to `0.0`.

neq

`neq(`*term*`,`*term*`)`
`neq(`*term*`,`*term*`,`*boolean*`|`*variable*`)`

This *primitive* will evaluate to a *truth value* of `1.0` if its two *terms* do not unify, and `0.0` if they do. It is a *short hand* to the `are.different` primitive. When used with three *terms*, the *primitive* will always evaluate to a *truth value* of `1.0` if its third *term* unify with the boolean value coming from the succes of the unification of the 2 first *terms*. For example:

```
?- neq(3,5,:e)
-> ( true ) := 1.00 (0.000) 1
?- neq(3,3,:e)
-> ( false ) := 1.00 (0.000) 1
```

5.4 Binary

All *primitives* related to handling *binary terms* are grouped in this category.

bin.load

`bin.load(`*string*`,`*binary*`|`*variable*`)`

This *primitive* will unifies or substitues its last *term* with a *binary* term which content was loaded from the file specified in the first term. If the load fails, the *primitive* will evaluate to a *truth value* of `0`. For example:

```
?- bin.load("test.bin",:b)
-> ( 'binary("aGVsbG8sIHdvcmxkIQA=") ) := 1.00 (0.000) 1
```

bin.length

`bin.length(`*binary*`,`*number*`|`*variable*`)`

This *primitive* will unifies or substitues its last *term* with the bytes size of its first *term*, a *binary*. If the first *term* isn't a *binary*, the *primitive* will evaluate to a *truth value* of `0`. For example:

```
?- bin.length('binary("aGVsbG8sIHdvcmxkIQA="),:l)
-> ( 14 ) := 1.00 (0.000) 1
```

bin.save

`bin.save(`*binary*`,`*string*`)`

This *primitive* will save it's first *term*, a *binary* into a file specified in the second term. If the save fails, the *primitive* will evaluate to a *truth value* of `0`. For example:

```
?- bin.save('binary("aGVsbG8sIHdvcmxkIQA="),"test.bin")
-> (  ) := 1.00 (0.000) 1
```

5.5 Data

All *primitives* related to handling *data terms* are grouped in this category.

daa.avg

`daa.avg(`*data*`,`*number*`|`*variable*`)`

This *primitive* will unifies or substitues the second *term* with the average of all values in the *data* (the first *term*). For example:

```
?- daa.make(byte,[5,12,245,56],:D), daa.avg(:D,:v)
-> ( 79.500000 ) := 1.00 (0.000) 1
```

daa.find

`daa.find(`*data*`,`*number*`|`*range*`,`*variable*`|`*term*`)`

The *primitive* `daa.find` will unifies or substitues its last *term* with a list of the indices of all the items in the *data* (the first *term*) which unifies with the second *term*, either a *number* or a *range*. For example:

```
?- daa.make(byte,[5,12,245,56],:D), daa.find(:D,<0|60>,:i)
-> ( [0, 1, 3] ) := 1.00 (0.000) 1
```

daa.format

`daa.format(`*data*`,`*symbol*`|`*variable*`)`

This *primitive* will unifies or substitues its last *term* with the format of the content in its first *term*, a *data*. If the first *term* isn't a *data*, the *primitive* will evaluate to a *truth value* of `0`. For example:

```
?- daa.make(byte,[5,12,245,56],:D), daa.format(:D,:f)
-> ( byte ) := 1.00 (0.000) 1
```

daa.item

`daa.item(`*data*`,`*number*`,`*variable*`|`*term*`)`

This *primitive* will unifies or substitues its last *term* with the nth item in first *term*, a *data*. If the index of the item (the second *term*) is outside of the bounds of the *data*, the *primitive* will evaluate to a *truth value* of `0`. For example:

```
?- daa.make(byte,[5,12,245,56],:D), daa.item(:D,0,:v)
-> ( 5 ) := 1.00 (0.000) 1
?- daa.make(byte,[5,12,245,56],:D), daa.item(:D,8,:v)
```

If the second *term* is an unbound *variable*, the *primitive* will enumerate all items in the *data* and their indices:

```
?- daa.make(byte,[5,12,245,56],:D), daa.item(:D,:i,:v)
-> ( 0 , 5 ) := 1.00 (0.000) 1
-> ( 1 , 12 ) := 1.00 (0.001) 2
-> ( 2 , 245 ) := 1.00 (0.001) 3
-> ( 3 , 56 ) := 1.00 (0.001) 4
```

If now only the second *term* is un unbound *variable*, the index of the first item unifying the third *term* will be bound:

```
?- daa.make(byte,[5,12,245,56],:D), daa.item(:D,:i,56)
-> ( 3 ) := 1.00 (0.000) 1
```

daa.length

`daa.length(`*data*`,`*number*`|`*variable*`)`

This *primitive* will unifies or substitues its last *term* with the number of items in its first *term*, a *data*. If the first *term* isn't a *data*, the *primitive* will evaluate to a *truth value* of `0`. For example:

```
?- daa.make(byte,[5,12,245,56],:D), daa.length(:D,:l)
-> ( 4 ) := 1.00 (0.000) 1
```

daa.make

`daa.make(`*symbol*`,`*list*`,`*term*`|`*variable*`)`

The *primitive* `daa.make` will unifies or substitues the last *term* with a new *data term*. The first *term*, a *symbol*, indicates the expected contents format while the second *term* is a *list* of the values to be stored in the *data* . For example:

```
?- daa.make(byte,[5,12,245,56],:d)
-> ( 'data(byte,"BQz1OA==") ) := 1.00 (0.000) 1
```

Supported content formats are: `byte`, `char`, `bool`, `uint16`, `sint16`, `uint32`, `sint32`, `uint64`, `sint64`, `real32` and `real64`.

daa.max

`daa.max(`*data*`,`*number*`|`*variable*`,`*number?*`|`*variable?*`)`

When only two *terms* are given, the *primitive* will unifies or substitues the second *term* with the value from the *data* (the first *term*) which have the highest value. If a third *term* is given, it will be unifies or substitues with the index of that value. For example:

```
?- daa.make(byte,[5,12,245,56],:D), daa.max(:D,:v)
-> ( 245 ) := 1.00 (0.000) 1
?- daa.make(byte,[5,12,245,56],:D), daa.max(:D,:v,:i)
-> ( 245 , 2 ) := 1.00 (0.000) 1
```

daa.member

`daa.member(`*term*`|`*variable*`,`*data*`)`

This *primitive* will unifies or substitues its first *term* with any of the items in its second *term*, a *data*. If the second *term* isn't a *data*, or if the unification of the 1st *term* fails, the *primitive* will evaluate to a *truth value* of `0`. For example:

```
?- daa.make(byte,[5,12,245,56],:D), daa.member(245,:D)
-> (  ) := 1.00 (0.000) 1
```

If the first *term* is an unbound *variable*, the *primitive* will enumerate all the items in the *data*:

```
?- daa.make(byte,[5,12,245,56],:D), daa.member(:i,:D)
-> ( 5 ) := 1.00 (0.000) 1
-> ( 12 ) := 1.00 (0.001) 2
-> ( 245 ) := 1.00 (0.001) 3
-> ( 56 ) := 1.00 (0.001) 4
```

daa.min

`daa.min(`*data*, *number*|*variable*, *number?*|*variable?*`)`

When only two *terms* are given, the *primitive* will unifies or substitues the second *term* with the value from the *data* (the first *term*) which have the lowest value. If a third *term* is given, it will be unifies or substitues with the index of that value. For example:

```
?- daa.make(byte,[5,12,245,56],:D), daa.min(:D,:v)
-> ( 5 ) := 1.00 (0.000) 1
?- daa.make(byte,[5,12,245,56],:D), daa.min(:D,:v,:i)
-> ( 5 , 0 ) := 1.00 (0.000) 1
```

5.6 Frame

All *primitives* related to handling *frames* are grouped in this category.

frm.erase

`frm.erase(`*frame*, *symbol*, *frame*|*variable*`)`

This *primitive* unifies or substitues the last *term* with the first *term* after the required label (the second a*terms*) have been removed in the *frame*. If the label isn't found in the *frame*, the *predicate* will still evaluate to `1.0`. For example:

```
?- frm.erase({a=4,b=5},a,:f)
-> ( {b = 5} ) := 1.00 (0.001) 1
?- frm.erase({a=4,b=5},c,:f)
-> ( {a = 4, b = 5} ) := 1.00 (0.000) 1
```

frm.fetch

`frm.fetch(`*frame*, *symbol*|*variable*, *term*|*variable*`)`
`frm.fetch(`*frame*, *symbol*|*variable*, *term*|*variable*, *term*`)`

The *primitive* `frm.fetch` main purpose is to get the value stored in a frame (the first *term*) for a given label (the second *term*) and unify it with the third *term*. If a fourth *term* is provided, it is considered to be the default value to be used to unify with the third in case the label isn't found in the *frame*. For example:

```
?- frm.fetch({a = 3, b = hello},b,:v)
-> ( hello ) := 1.00 (0.000) 1
```

If the second *term* is an unbound variable, the inference engine will list all label/value combinations:

```
?- frm.fetch({a = 3, b = hello},:l,:v)
-> ( a , 3 ) := 1.00 (0.000) 1
-> ( b , hello ) := 1.00 (0.001) 2
```

frm.length

`frm.length(`*frame*`,`*number*`|`*variable*`)`

This *primitive* will unify or substitue its second *term* with the length (that is the number of items) in the *frame* passed as first *term*.

```
?- frm.length({a = 3, b = hello},:l)
-> ( 2 ) := 1.00 (0.000) 1
```

frm.make

`frm.make(`*list+*`,`*frame*`|`*variable*`)`

This *primitive* will unify or substitue its last *term* with a *frame* created from a collection of label/value pairs. For example:

```
?- frm.make([a,4],[b,"hello"],:f)
-> ( {a = 4, b = "hello"} ) := 1.00 (0.000) 1
?- frm.make([[a,4],[b,"hello"]],:f)
-> ( {a = 4, b = "hello"} ) := 1.00 (0.000) 1
```

frm.store

`frm.store(`*frame*`,`*symbol*`|`*variable*`,`*term*`,`*frame*`|`*variable*`)`

This *primitive* unifies or substitues the last *term* with the first *term* after the required label/value pair (the second and third *terms*) have been updated or inserted in the *frame*. For example:

```
?- frm.store({a = 3, b = hello},c,"world!",:o)
-> ( {a = 3, b = hello, c = "world!"} ) := 1.00 (0.000) 1
```

frm.empty

`frm.empty(`*frame*`)`

The *primitive* `frm.empty` will resolve with a *truth value* of `1` if its sole *term* is an empty *frame*. For example:

```
?- frm.empty({})
-> (  ) := 1.00 (0.000) 1
?- frm.empty({a = 1})
-> (  ) := 0.00 (0.001) 1
```

frm.label

`frm.label(`*frame*`,`*symbol*`|`*variable*`)`

With this *primitive*, it is possible to check if a given *label* exists in the *frame*. It will resolve with a *truth value* of `1` if the *label* exists. `0`, otherwise:

```
?- frm.label({a=1,b=2,c=3},a)
-> (  ) := 1.00 (0.000) 1
?- frm.label({a=1,b=2,c=3},d)
-> (  ) := 0.00 (0.000) 1
```

If the second *term* is an unbound variable, the *inference* will generate as many solutions as there are pairs in the *frame*:

```
?- frm.label({a=1,b=2,c=3},:label)
-> ( a ) := 1.00 (0.000) 1
-> ( b ) := 1.00 (0.000) 2
-> ( c ) := 1.00 (0.000) 3
```

frm.labels

`frm.labels(`*frame*`,`*list*`|`*variable*`)`

This *primitive* will unify or substitue its second *term* with a list of all the labels of the label/value pairs in the *frame*.

```
?- frm.labels({a=1,b=2,c=3},:labels)
-> ( [a, b, c] ) := 1.00 (0.000) 1
```

When the second *term* is a *list* of *symbols*, the list ordering doesn't have to match the order in which the *frame* label/value pairs have been specified:

```
?- frm.labels({a=1,b=2,c=3},[a,b,c])
-> (  ) := 1.00 (0.000) 1
?- frm.labels({a=1,b=2,c=3},[b,a,c])
-> (  ) := 1.00 (0.001) 1
?- frm.labels({a=1,b=2,c=3},[b,d,a])
-> (  ) := 0.00 (0.000) 1
?- frm.labels({a=1,b=2,c=3},[b,c,a])
-> (  ) := 1.00 (0.000) 1
```

frm.values

`frm.values(`*frame*`,`*list*`|`*variable*`)`

This *primitive* will unify or substitue its second *term* with a *list* of all the values of the label/value pairs in the *frame*.

```
?- frm.values({a=1,b=2,c=3},:labels)
-> ( [1, 2, 3] ) := 1.00 (0.000) 1
```

Just like with the `frm.labels` *primitive*, the *list* ordering doesn't have to match:

```
?- frm.values({a=1,b=2,c=3},[1,2,3])
-> (  ) := 1.00 (0.000) 1
?- frm.values({a=1,b=2,c=3},[1,3,2])
-> (  ) := 1.00 (0.000) 1
?- frm.values({a=1,b=2,c=3},[1,3,4])
-> (  ) := 0.00 (0.000) 1
```

frm.pairs

`frm.pairs(`*frame*`,`*list*`|`*variable*`)`

The *primitive* `frm.pairs` will unify or substitue its second *term* with a *list* of all the label/value pairs in the *frame*. Each of the pairs will be stored in a *list* of two elements as seen in this example:

```
?- frm.pairs({a=1,b=2,c=3},:pairs)
-> ( [[a, 1], [b, 2], [c, 3]] ) := 1.00 (0.000) 1
```

When the second *term* is a *list* that contains *lists*, the *list* ordering doesn't have to match the order in which the *frame* label/value pairs have been specified.

frm.cat

`frm.cat(`*frame*`,`*frame*`,`*frame*`|`*variable*`)`

Thie *primitive* will merge two *frames* and unify/substitute it with the third *term*.

```
?- frm.cat({a=1,b=2,c=3},{d=4},:merged)
-> ( {a = 1, b = 2, c = 3, d = 4} ) := 1.00 (0.000) 1
```

When a label exists in both *frames*, both value will be put in a *list* and the *list* will be stored in the output *frame*:

```
?- frm.cat({a=1,b=2,c=3},{c=4},:merged)
-> ( {a = 1, b = 2, c = [3, 4]} ) := 1.00 (0.000) 1
```

frm.sub

`frm.sub(`*frame*`,`*list*`,`*frame*`|`*variable*`)`

This *primitive* will extract a collection of label/value pairs from the *frame* given as the first *term* and unify or substitue its third *term* with a *frame* containing them. The second *term* is a list of all the labels to be included. Here's an example:

```
?- frm.sub({a=1,b=2,c=3},[a,c],:sub)
-> ( {a = 1, c = 3} ) := 1.00 (0.000) 1
```

frm.swap

`frm.swap(`*frame*`,`*frame*`,`*frame*`|`*variable*`)`

This *primitive* unifies or substitues the last *term* with a new *frame* containing the concatenation of its first two *terms*. If a label in the second *frame* is already present in the first *frame*, its value will be replaced by the one from the second *term*. For example:

```
?- frm.swap({a=4,b=2},{a=3,c=5},:f)
-> ( {a = 3, b = 2, c = 5} ) := 1.00 (0.000) 1
```

frm.there

`frm.there(`*frame*`,`*frame*`)`

With this *primitive*, it is possible to check if all *labels* in the first *term* exists in the second *term frame*. It will resolve with a *truth value* of `1` if so. `0`, otherwise:

```
?- frm.there({a = _, b = _},{c = 4})
-> (  ) := 0.00 (0.000) 1
?- frm.there({a = _, b = _},{a = 4})
-> (  ) := 0.00 (0.000) 1
?- frm.there({a = _, b = _},{a = 4, b = 3})
-> (  ) := 1.00 (0.000) 1
?- frm.there({a = _, b = _},{a = 4, b = 3, c = 3})
-> (  ) := 1.00 (0.000) 1
```

frm.same

`frm.same(`*frame*`,`*frame*`)`

The *primitive* `frm.same` will resolve with a *truth value* of `1` if both *terms* are identical *frames*. For example:

```
?- frm.same({a=3},{b=3})
-> (  ) := 0.00 (0.000) 1
?- frm.same({a=3},{a=3})
-> (  ) := 1.00 (0.000) 1
?- frm.same({a=3},{a=3,b=2})
-> (  ) := 0.00 (0.000) 1
```

5.7 Functor

This section covers all the *primitives* that manipulate *functors*.

fun.length

`fun.length(`*functor*`,`*number*`|`*variable*`)`

This *primitive* will unify or substitue its second *term* with the length (that is, the *arity*) of the *functor* passed as first *term*.

```
?- fun.length(truck(red,1930,ford),:l)
-> ( 3 ) := 1.00 (0.000) 1
```

fun.make

`fun.make(`*symbol*`,`*list*`,`*functor*`|`*variable*`)`

The `fun.make` *primitive* unify or substitue its third *term* with a *functor* created from the first (the label) and second (the list of *terms*) *terms*. For example:

```
?- fun.make(product,[\:name,apple,_],:func)
-> ( product(:name, apple, _) ) := 1.00 (0.000) 1
```

fun.member

`fun.member(`*functor*`,`*term*`)`

This *primitive* will resolve to the *truth value* of `1` only if the second *term* unifies with any of the *terms* in the *functor*. For example:

```
?- fun.member(truck(red,1930,ford),ford)
-> (  ) := 1.00 (0.000) 1
?- fun.member(truck(red,1930,ford),red)
-> (  ) := 1.00 (0.000) 1
?- fun.member(truck(red,1930,ford),green)
-> (  ) := 0.00 (0.000) 1
```

If the second *term* is an unbound *variable*, the *primitive* will generate as many *statements* as there are *terms* in the *functor*:

```
?- fun.member(truck(red,1930,ford),:x)
-> ( red ) := 1.00 (0.000) 1
-> ( 1930 ) := 1.00 (0.000) 2
-> ( ford ) := 1.00 (0.000) 3
```

fun.label

`fun.label(`*functor*`,`*symbol*`|`*variable*`)`

This *primitive* will unify or substitue its second *term* with the label of the *functor* passed as first *term*.

fun.terms

`fun.terms(`*functor*`,`*list*`|`*variable*`)`

This *primitive* will unify or substitue its second *term* with a list of the *functor*'s terms. For example:

```
?- fun.terms(truck(red,1930,ford),:terms)
-> ( [red, 1930, ford] ) := 1.00 (0.002) 1
```

When the second *term* is a *list*, it will have to be ordered the same way to successfully unify.

5.8 List

All *primitives* related to handling *lists* are grouped in this category.

lst.all

`lst.all(`*list*`,`*list*`)`

The *primitive* `lst.all` will resolve with a *truth value* of `1` if all the element in its second *term* (a *list*) are found in its first *term*, also a *list*. For example:

```
?- lst.all([a,b,c,d,e],[b,d])
-> (  ) := 1.00 (0.000) 1
?- lst.all([a,b,c,d,e],[b,d,f])
-> (  ) := 0.00 (0.000) 1
```

lst.any

`lst.any(`*list*`,`*list*`)`

The *primitive* `lst.all` will resolve with a *truth value* of `1` if any of the element in its second *term* (a *list*) is found in its first *term*, also a *list*. For example:

```
?- lst.any([a,b,c,d,e],[b,d,f])
-> (  ) := 1.00 (0.000) 1
?- lst.any([a,b,c,d,e],[f])
-> (  ) := 0.00 (0.000) 1
```

lst.avg

`lst.avg(`*list*`,`*term*`|`*variable*`)`

This *primitive* will unify its second *term* with the computed average of all elements in the *list* given as first *term*. For example:

```
?- lst.avg([1,5,0,8],:v)
-> ( 3.500000 ) := 1.00 (0.000) 1
```

lst.bulk

`lst.bulk(`*list*`,`*list*`|`*variable*`)`

This *primitive* will unify or substitue its second *term* with the *list* containing all elements but the last one, in the *list* passed as first *term*:

```
?- lst.bulk([a,b,c,d],:h)
-> ( [a, b, c] ) := 1.00 (0.000) 1
```

If a 3rd term is provided, it will be unified with the tail element in the *list*.

lst.cat

`lst.cat(`*term+*`,`*list*`|`*variable*`)`

The *primitive* unifies the last *term* with a concatenation of all the other *terms* into a *list*. For example:

```
?- lst.cat(1,2,3,4,:l)
-> ( [1, 2, 3, 4] ) := 1.00 (0.000) 1
?- lst.cat([1,2],3,[4],:l)
-> ( [1, 2, 3, 4] ) := 1.00 (0.000) 1
```

lst.combi

`lst.combi(`*list*`,`*list*`|`*variable*`)`

The `lst.combi` *primitive* will unify its second *term* with every possible combination of the elements in the first *term* (expected to be a *list*). For example:

```
?- lst.combi([a,b,c,d],:l)
-> ( [] ) := 1.00 (0.000) 1
-> ( [a] ) := 1.00 (0.001) 2
-> ( [b] ) := 1.00 (0.001) 3
-> ( [c] ) := 1.00 (0.001) 4
-> ( [d] ) := 1.00 (0.001) 5
-> ( [a, b] ) := 1.00 (0.001) 6
-> ( [a, c] ) := 1.00 (0.001) 7
-> ( [a, d] ) := 1.00 (0.001) 8
-> ( [b, c] ) := 1.00 (0.001) 9
-> ( [b, d] ) := 1.00 (0.001) 10
-> ( [c, d] ) := 1.00 (0.001) 11
-> ( [a, b, c] ) := 1.00 (0.001) 12
-> ( [a, b, d] ) := 1.00 (0.001) 13
-> ( [a, c, d] ) := 1.00 (0.001) 14
-> ( [b, c, d] ) := 1.00 (0.002) 15
-> ( [a, b, c, d] ) := 1.00 (0.002) 16
```

lst.diff

`lst.diff(`*list*`)`

The *primitive* `lst.diff` will resolve with a *truth value* of `1` if its sole *term* is a *list* whose elements are all unique. For example:

```
?- lst.diff([a,b,c,d])
-> (  ) := 1.00 (0.000) 1
?- lst.diff([a,b,a,d])
-> (  ) := 0.00 (0.000) 1
```

lst.empty

`lst.empty(`*list*`)`

The *primitive* `lst.empty` will resolve with a *truth value* of `1` if its sole *term* is an empty *list*. For example:

```
?- lst.empty([a,b,c,d])
-> (  ) := 0.00 (0.000) 1
?- lst.empty([])
-> (  ) := 1.00 (0.000) 1
```

lst.except

`lst.except(`*term*,*list*`)`

The `lst.except` *primitive* will resolve to a *truth value* of `1.0` if its first *term* is not in the list provided as second *term*, like in the following example:

```
?- lst.except(3,[3,2])
-> (  ) := 0.00 (0.000) 1
?- lst.except(5,[3,2])
-> (  ) := 1.00 (0.000) 1
```

lst.excl

`lst.excl(`*list*`,`*list*`)`

The `lst.excl` *primitive* will resolve to a *truth value* of `1.0` if all *terms* in its second *term* are not present in the *list* given as first *term*. For example:

```
?- lst.excl([a,b,c,d],[c,b])
-> (  ) := 0.00 (0.000) 1
?- lst.excl([a,b,c,d],[e,f])
-> (  ) := 1.00 (0.000) 1
```

lst.find

`lst.find(`*list*`,`*term*`,`*list*`|,`*variable*`)`

The `lst.find` *primitive* will unify its third *term* with a list of the indices in the *list* at which the second *term* is present. For example:

```
?- lst.find([a,b,c,d],a,:v)
-> ( [0] ) := 1.00 (0.000) 1
?- lst.find([a,b,a,d],a,:v)
-> ( [0, 2] ) := 1.00 (0.000) 1
?- lst.find([a,b,a,d],e,:v)
-> ( [] ) := 1.00 (0.000) 1
```

lst.flat

`lst.flat(`*list*`,`*list*`|`*variable*`)`

The `lst.flat` *primitive* will unify its second *term* with a *list* whose content is the first *term* flattened. For example:

```
?- lst.flat([a,[b,[c,d]],e,[f,[g]]],:l)
-> ( [a, b, c, d, e, f, g] ) := 1.00 (0.000) 1
```

lst.flip

`lst.flip(`*list*`|`*variable*`,`*list*`|`*variable*`)`

The `lst.flip` *primitive* will unify both *terms* with a *list* whose content is the inverse of the content of whichever *term* is a *list*. For example:

```
?- lst.flip([a,b,c,d],:l)
-> ( [d, c, b, a] ) := 1.00 (0.000) 1
?- lst.flip(:l,[a,b,c,d])
-> ( [d, c, b, a] ) := 1.00 (0.000) 1
```

lst.head

`lst.head(`*list*`,`*term*`)`

This *primitive* will unify or substitue its second *term* with the head (the first element)) in the *list* passed as first *term*:

```
?- lst.head([a,b,c,d],:h)
-> ( a ) := 1.00 (0.000) 1
```

lst.incl

`lst.incl`(*list*, *list*)

The `lst.incl` *primitive* will resolve to a *truth value* of `1.0` if all *terms* in its second *term* are present in the *list* given as first *term*. For example:

```
?- lst.incl([a,b,c,d],[c,b])
-> (  ) := 1.00 (0.000) 1
?- lst.incl([a,b,c,d],[e,f])
-> (  ) := 0.00 (0.000) 1
```

lst.init

`lst.init`(*list*, *list*|*variable*)

The `lst.init` *primitive* will unify its second *term* with a *list* containing all the items from the *list* given as first *term* but the last item. For example:

```
?- lst.init([a,b,c,d,e],:l)
-> ( [a, b, c, d] ) := 1.00 (0.001) 1
```

lst.it

`lst.it`(*term*, *term*|*variable*)

This *primitive* will unify its second *term* with either the first *term* if it's a *list*, or a *list* containing the first *term* if it isn't. For example:

```
?- lst.it(a,:l)
-> ( [a] ) := 1.00 (0.000) 1
?- lst.it([a],:l)
-> ( [a] ) := 1.00 (0.000) 1
```

lst.item

`lst.item`(*list*, *number*|*variable*, *term*|*variable*)

This *primitive* can be used to get a given element from a *list* based on its index, or find the index of the first occurence of a *term* in the *list*:

```
?- lst.item([a,b,c,d],0,:e)
-> ( a ) := 1.00 (0.000) 1
?- lst.item([a,b,c,d],:i,b)
-> ( 1 ) := 1.00 (0.000) 1
```

When the last two *terms* of the *primitive* are unbound *variables*, it will generate all possible combinations of the two *terms*:

```
?- lst.item([a,b,c,d],:i,:v)
-> ( 0 , a ) := 1.00 (0.000) 1
-> ( 1 , b ) := 1.00 (0.001) 2
-> ( 2 , c ) := 1.00 (0.001) 3
-> ( 3 , d ) := 1.00 (0.001) 4
```

lst.join

`lst.join`(*list*, *list*, *list*| *variable*)

The `lst.join` *primitive* will combine the content of its first two *terms* (without duplicates) into a *list* to be unified with the third *term*. For example:

```
?- lst.join([a,b,c,d],[d,e,f],:l)
-> ( [a, d, b, e, c, f] ) := 1.00 (0.000) 1
```

lst.knit

`lst.knit`(*list+*, *list*| *variable*)

This *primitive* will interleave the elements of each *lists* given as argument into a list and unify it with the last *term*. For example:

```
?- lst.knit([1,2,3,4],[a,b,c,d],:l)
-> ( [[1, a], [2, b], [3, c], [4, d]] ) := 1.00 (0.000) 1
?- lst.knit([1,2,3,4],[a,b,c],[e,f],:l)
-> ( [[1, a, e], [2, b, f]] ) := 1.00 (0.000) 1
```

lst.length

`lst.length`(*list*, *number*| *variable*)
`lst.length`(*variable*, *number*, *term?*)

This *primitive* will unify or substitue its second *term* with the length (that is the number of items) in the *list* passed as first *term*.

```
?- lst.length([1,2,3,4,5],:l)
-> ( 5 ) := 1.00 (0.000) 1
```

If the first *term* is an unbound *variable* and the second *term* is a *number*, the *variable* will be bound to a list of that size filled with *wilcard variable*:

```
?- lst.length(:l,5)
-> ( [_, _, _, _, _] ) := 1.00 (0.000) 1
```

An optional third *term* can be given when a *list* is being created to be the *term* to be used to fill the *list* instead of the *wilcard variable*. For example:

```
?- lst.length(:l,5,0)
-> ( [0, 0, 0, 0, 0] ) := 1.00 (0.000) 1
```

lst.make

`lst.make(`*term+*`,`*list*`|`*variable*`)`

This *primitive* unifies the last *term* with a *list* containing all the other *terms*. For example:

```
?- lst.make([a],b,c,d,:l)
-> ( [[a], b, c, d] ) := 1.00 (0.001) 1
?- lst.make(a,b,c,d,:l)
-> ( [a, b, c, d] ) := 1.00 (0.001) 1
```

lst.max

`lst.max(`*list*`,`*term*`|`*variable*`)`

This *primitive* will unify its second *term* with the maximum value of all elements in the list given as first *term*. For example:

```
?- lst.max([1,5,0,8],:l)
-> ( 8 ) := 1.00 (0.000) 1
?- lst.max([1,5,0,8],8)
-> (  ) := 1.00 (0.000) 1
?- lst.max([1,5,0,8],9)
-> (  ) := 0.00 (0.000) 1
```

lst.member

`lst.member(`*term*`|`*variable*`,`*list*`|`*variable*`)`

The `lst.member` *primitive* will unify the first *term* with each element of the list provided as second *term*, like in the following example:

```
?- lst.member(:x,[3,2])
-> ( 3 ) := 1.00 (0.000) 1
-> ( 2 ) := 1.00 (0.000) 2
?- lst.member(3,[3,2])
-> (  ) := 1.00 (0.000) 1
?- lst.member(5,[3,2])
-> (  ) := 0.00 (0.000) 1
```

The *primitive* can be used to generate all possible combinations when used with a *list* having *wildcard variables* in it. Here's an example:

```
?- set(:l,[a,_,c,_,e]), lst.member(f,:l), lst.member(g,:l)
-> ( [a, f, c, g, e] ) := 1.00 (0.001) 1
-> ( [a, g, c, f, e] ) := 1.00 (0.001) 2
```

lst.min

`lst.min(`*list*`,`*term*`|`*variable*`)`

This *primitive* will unify its second *term* with the minimum value of all elements in the list given as first *term*. For example:

```
?- lst.min([1,5,0,8],:l)
-> ( 0 ) := 1.00 (0.000) 1
?- lst.min([1,5,0,8],0)
-> (  ) := 1.00 (0.000) 1
?- lst.min([1,5,0,8],9)
-> (  ) := 0.00 (0.000) 1
```

lst.mix

`lst.mix`(*list*, *list*| *variable*)

This *primitive* will unify or bind its second *term* with a copy of its first *term* where the elements have been scrambled randomly within the *list*. For example:

```
?- lst.mix([1,2,3,4,5,6,7,8,9,0],:l)
-> ( [9, 1, 2, 8, 7, 5, 3, 0, 6, 4] ) := 1.00 (0.001) 1
?- lst.mix([1,2,3,4,5,6,7,8,9,0],:l)
-> ( [2, 6, 7, 0, 1, 9, 5, 3, 8, 4] ) := 1.00 (0.001) 1
```

lst.permu

`lst.permu`(*list*, *list*| *variable*)

The `lst.permu` *primitive* will unify its second *term* with every possible permutations of the elements in the first *term* (expected to be a *list*). For example:

```
?- lst.permu([a,b,c,d],:l)
-> ( [a, b, c, d] ) := 1.00 (0.001) 1
-> ( [a, b, d, c] ) := 1.00 (0.002) 2
-> ( [a, c, b, d] ) := 1.00 (0.002) 3
-> ( [a, c, d, b] ) := 1.00 (0.002) 4
-> ( [a, d, c, b] ) := 1.00 (0.002) 5
-> ( [a, d, b, c] ) := 1.00 (0.002) 6
-> ( [b, a, c, d] ) := 1.00 (0.002) 7
-> ( [b, a, d, c] ) := 1.00 (0.002) 8
-> ( [b, c, a, d] ) := 1.00 (0.002) 9
-> ( [b, c, d, a] ) := 1.00 (0.002) 10
-> ( [b, d, c, a] ) := 1.00 (0.002) 11
-> ( [b, d, a, c] ) := 1.00 (0.002) 12
-> ( [c, b, a, d] ) := 1.00 (0.002) 13
-> ( [c, b, d, a] ) := 1.00 (0.002) 14
-> ( [c, a, b, d] ) := 1.00 (0.002) 15
-> ( [c, a, d, b] ) := 1.00 (0.003) 16
-> ( [c, d, a, b] ) := 1.00 (0.003) 17
-> ( [c, d, b, a] ) := 1.00 (0.003) 18
-> ( [d, b, c, a] ) := 1.00 (0.003) 19
-> ( [d, b, a, c] ) := 1.00 (0.003) 20
-> ( [d, c, b, a] ) := 1.00 (0.003) 21
-> ( [d, c, a, b] ) := 1.00 (0.003) 22
-> ( [d, a, c, b] ) := 1.00 (0.003) 23
-> ( [d, a, b, c] ) := 1.00 (0.003) 24
```

lst.remove

`lst.remove`(*term*, *list*, *list*| *variable*)

The `lst.remove` *primitive* will resolve to a *truth value* of `1.0` if its first *term* is in the list provided as second *term*, and will unify or substitue its third *term* with a copy of its second *term* where all instances of the first *term* as been removed. For example:

```
?- lst.remove(a,[a,b,c,a,d],:l)
-> ( [b, c, d] ) := 1.00 (0.000) 1
```

If the first *term* is a string and that a fourth *term* is provided, it will be considered to be a normalized Levenshtein threshold that will be used to compare the first *term* against all other strings in the list.

lst.rest

`lst.rest(`*list*, *list*| *variable*`)`

This *primitive* will unify or substitue its second *term* with the tail (a *list* containing all elements but the first) in the *list* passed as first *term*:

```
?- lst.rest([a,b,c,d],:h)
-> ( [b, c, d] ) := 1.00 (0.000) 1
```

lst.snap

`lst.snap(`*list*, *term*, *list*| *variable*, *list*| *variable*`)`

The `lst.snap` *primitive* will unify its third and fourth *terms* each with a *list* that contains the elements of the first *term* split according to the first occurence of the second *term* in the *list*. For example:

```
?- lst.snap([a,b,c,d,e,f],d,:h,:t)
-> ( [a, b, c] , [e, f] ) := 1.00 (0.000) 1
?- lst.snap([a,b,c,d,e,f],a,:h,:t)
-> ( [] , [b, c, d, e, f] ) := 1.00 (0.000) 1
?- lst.snap([a,b,c,d,e,f],f,:h,:t)
-> ( [a, b, c, d, e] , [] ) := 1.00 (0.000) 1
?- lst.snap([a,b,c,d,e,f],g,:h,:t)
```

If the first *term* is a *variable*, and the third and fourth *terms* are *lists*, a concatenation of both list with the second *term* will be done:

```
?- lst.snap(:l,d,[a,b,c],[e,f,g])
-> ( [a, b, c, d, e, f, g] ) := 1.00 (0.000) 1
```

lst.span

`lst.span(`*range*| *list*, *list*`)`

This *primitive* will unify a *range* (first term) over all the elements of a *list* without having the same element twice in the output *list* (the third *term*). For example:

```
?- lst.length(:l,4), lst.span(<1|4>,:l);
-> ( [1, 2, 3, 4] ) := 1.00 (0.001) 1
-> ( [1, 2, 4, 3] ) := 1.00 (0.001) 2
-> ( [1, 3, 2, 4] ) := 1.00 (0.001) 3
-> ( [1, 3, 4, 2] ) := 1.00 (0.001) 4
-> ( [1, 4, 3, 2] ) := 1.00 (0.001) 5
```

```
-> ( [1, 4, 2, 3] ) := 1.00 (0.001) 6
-> ( [2, 1, 3, 4] ) := 1.00 (0.001) 7
-> ( [2, 1, 4, 3] ) := 1.00 (0.001) 8
-> ( [2, 3, 1, 4] ) := 1.00 (0.001) 9
-> ( [2, 3, 4, 1] ) := 1.00 (0.001) 10
-> ( [2, 4, 3, 1] ) := 1.00 (0.001) 11
-> ( [2, 4, 1, 3] ) := 1.00 (0.001) 12
-> ( [3, 2, 1, 4] ) := 1.00 (0.001) 13
-> ( [3, 2, 4, 1] ) := 1.00 (0.002) 14
-> ( [3, 1, 2, 4] ) := 1.00 (0.002) 15
-> ( [3, 1, 4, 2] ) := 1.00 (0.002) 16
-> ( [3, 4, 1, 2] ) := 1.00 (0.002) 17
-> ( [3, 4, 2, 1] ) := 1.00 (0.002) 18
-> ( [4, 2, 3, 1] ) := 1.00 (0.002) 19
-> ( [4, 2, 1, 3] ) := 1.00 (0.002) 20
-> ( [4, 3, 2, 1] ) := 1.00 (0.002) 21
-> ( [4, 3, 1, 2] ) := 1.00 (0.002) 22
-> ( [4, 1, 3, 2] ) := 1.00 (0.002) 23
-> ( [4, 1, 2, 3] ) := 1.00 (0.002) 24
?- lst.length(:l,3), lst.span([a,b,c],:l);
-> ( [a, b, c] ) := 1.00 (0.000) 1
-> ( [a, c, b] ) := 1.00 (0.001) 2
-> ( [b, a, c] ) := 1.00 (0.001) 3
-> ( [b, c, a] ) := 1.00 (0.001) 4
-> ( [c, b, a] ) := 1.00 (0.001) 5
-> ( [c, a, b] ) := 1.00 (0.001) 6
```

lst.sort

`lst.sort(`*list*, *list*`)`
`lst.sort(`*list*, *list*, *number*`)`

This *primitive* will unify or bind its last *term* with a copy of its first *term* where the elements have been sorted in increasing order. If a third *term* is given, it will be assumed that the *list* to sort contains *lists* and that the number is the index of the element to be used for sorting the *lists*. For example:

```
?- lst.sort([3,7,1,9,4,3],:l)
-> ( [1, 3, 3, 4, 7, 9] ) := 1.00 (0.001) 1
?- lst.sort([[3,a],[7,b],[1,d],[9,f],[4,e],[3,z]],:l,1)
-> ( [[3, a], [7, b], [1, d], [4, e], [9, f], [3, z]] ) := 1.00 (0.001) 1
```

Only *atoms* and *lists* (when a third *term* is given) can be sorted.

lst.split

`lst.split(`*list*, *list*| *variable*, *list*| *variable*`)`

This *primitive* will unify or bind its second and third *terms* with every possible split of the first term (a *list*).For example:

```
?- lst.split([a,b,c,d],:l,:r)
-> ( [] , [a, b, c, d] ) := 1.00 (0.000) 1
-> ( [a] , [b, c, d] ) := 1.00 (0.001) 2
-> ( [a, b] , [c, d] ) := 1.00 (0.001) 3
-> ( [a, b, c] , [d] ) := 1.00 (0.001) 4
-> ( [a, b, c, d] , [] ) := 1.00 (0.001) 5
```

If the first term is an unbound *variable* and the two other *terms* are *lists*, the *primitive* will unify the first *term* with a *list* concatenating both *lists*. For example:

```
?- lst.split(:l,[a,b,c],[d])
-> ( [a, b, c, d] ) := 1.00 (0.000) 1
```

lst.sub

`lst.sub(`*list*|*variable*, *number*, *number*, *list*|*variable*`)`

The `lst.sub` *primitive* will unify or substitue its fourth *term* with a subpart of the *list* given as first *term*. The subpart is defined by an offset (second *term*) and a length (third *term*). For example:

```
?- lst.sub([1,2,3,4,5,6],4,2,[5,:x])
-> ( 6 ) := 1.00 (0.000) 1
```

If the first and fourth *terms* are both *lists* and the offset is a un-bound *variable*, the call will unify the offset will possible occurences of the fourth *term* in the list. As in this example:

```
?- lst.sub([1,2,3,4,5,6,8,5,6],:i,:v,[5,6])
-> ( 4 , 2 ) := 1.00 (0.001) 1
-> ( 7 , 2 ) := 1.00 (0.001) 2
```

lst.swap

`lst.swap(`*list*, *number*, *term*, *variable*|*list*`)`

This *primitive* will unify or bind its last *term* with a copy of its first *term* where the element at the position given as second *term* has been swapped for the third *term*. For example:

```
?- lst.swap([a,b,c,d,e],0,f,:l)
-> ( [f, b, c, d, e] ) := 1.00 (0.001) 1
?- lst.swap([a,b,c,d,e],3,f,:l)
-> ( [a, b, c, f, e] ) := 1.00 (0.001) 1
```

lst.tail

`lst.tail(`*list*, *list*|*variable*`)`

This *primitive* will unify or substitue its second *term* with the tail (the last element) in the *list* passed as first *term*:

```
?- lst.tail([a,b,c,d],:h)
-> ( d ) := 1.00 (0.000) 1
```

lst.unique

`lst.unique(`*list*, *list*|*variable*`)`

This *primitive* will unify or bind its second *term* with a copy of its first *term* where all duplicated elements in the *list* have been removed. For example:

```
?- lst.unique([a,b,c,d,a,e,a,f,b,g],:l)
-> ( [a, b, c, d, e, f, g] ) := 1.00 (0.000) 1
```

5.9 Boolean Logic

This section contains all the *primitives* that deal with *boolean logic* operations.

boo.and

`boo.and(`*boolean+*`,`*boolean*`|`*variable*`)`

This *primitive* will unify or bind its last *term* with the boolean `AND` of all other *terms*. For example:

```
?- boo.and(1,0,1,:v)
-> ( false ) := 1.00 (0.000) 1
?- boo.and(1,1,1,:v)
-> ( true ) := 1.00 (0.000) 1
```

boo.not

`boo.not(`*boolean*`|`*variable*`,`*boolean*`|`*variable*`)`

This *primitive* will unify or bind its *terms* with the boolean negation of the other *term*. For example:

```
?- boo.not(1,:v)
-> ( false ) := 1.00 (0.001) 1
?- boo.not(0,:v)
-> ( true ) := 1.00 (0.000) 1
?- boo.not(0,1)
-> (  ) := 1.00 (0.001) 1
?- boo.not(:v,1)
-> ( false ) := 1.00 (0.000) 1
```

boo.or

`boo.or(`*boolean+*`,`*boolean*`|`*variable*`)`

This *primitive* will unify or bind its last *term* with the boolean `OR` of all other *terms*. For example:

```
?- boo.or(1,0,1,:v)
-> ( true ) := 1.00 (0.000) 1
?- boo.or(1,1,1,:v)
-> ( true ) := 1.00 (0.000) 1
?- boo.or(0,0,:v)
-> ( false ) := 1.00 (0.000) 1
```

boo.xor

`boo.xor(`*boolean+*`,`*boolean*`|`*variable*`)`

This *primitive* will unify or bind its last *term* with the boolean *exclusive disjunction* of all other *terms*. For example:

```
?- boo.xor(1,1,:v)
-> ( false ) := 1.00 (0.001) 1
?- boo.xor(1,0,:v)
-> ( true ) := 1.00 (0.001) 1
?- boo.xor(1,0,1,:v)
```

```
-> ( false ) := 1.00 (0.001) 1
?- boo.xor(1,0,0,:v)
-> ( true ) := 1.00 (0.001) 1
```

5.10 Mathematics

This section contains all the *primitives* that deal with *mathematical* operations.

mao.abs

`mao.abs(`*number*`|`*variable*`,`*number*`|`*variable*`)`

This *primitive* will unify or bind the second *term* with the absolute value of the first *term*. If the second *term* is a *number* and the first one is an unbound *variable* the call will generates two *statements*. For example:

```
?- mao.abs(2,:v)
-> ( 2 ) := 1.00 (0.000) 1
?- mao.abs(-2,:v)
-> ( 2 ) := 1.00 (0.000) 1
?- mao.abs(:v,4)
-> ( -4 ) := 1.00 (0.000) 1
-> ( 4 ) := 1.00 (0.000) 2
```

mao.atan2

`mao.atan2(`*number*`,`*number*`,`*number*`|`*variable*`)`

The *primitive* `mao.atan2` will unify or bind its third *term* with the principal value of the arc tangent of its first *term* divided by its second, expressed in degrees. For example:

```
?- mao.atan2(10,-10,:v)
-> ( 135 ) := 1.00 (0.000) 1
```

mao.ceil

`mao.ceil(`*number*`|`*variable*`,`*number*`|`*variable*`)`

This *primitive* will unify or bind the second *term* with the smallest integer value greater than or equal to the first *term*. For example:

```
?- mao.ceil(2.1,:x)
-> ( 3 ) := 1.00 (0.000) 1
?- mao.ceil(2.5,:x)
-> ( 3 ) := 1.00 (0.000) 1
?- mao.ceil(2.99,:x)
-> ( 3 ) := 1.00 (0.000) 1
```

If the second *term* is a *number* and the first one is an unbound *variable*, the *primitive* will bind the *variable* with a *range* value:

```
?- mao.ceil(:r,3)
-> ( <2.000001|2.999999> ) := 1.00 (0.000) 1
```

mao.cos

`mao.cos(`*number*`,`*number*`|`*variable*`)`

The *primitive* `mao.cos` will unify or bind its second *term* with the cosine of the angle (in degrees) value given as first *term*. If the first *term* is an unbound *variable* and the second is a *number* then the *primitive* will unify the first *term* with the arc-cosine. For example:

```
?- mao.cos(60,:v)
-> ( 0.500000 ) := 1.00 (0.000) 1
?- mao.cos(:a,0.5)
-> ( 60.000000 ) := 1.00 (0.000) 1
```

mao.d2r

`mao.d2r(`*number*`|`*variable*`,`*number*`|`*variable*`)`

This *primitive* will unify or bind its second *term* with the conversion from degrees to radians of the first *term*. If the first *term* is an unbound *variable* and the second *term* is a *number*, it will bind the *variable* with the conversion from radians to degree of the second *term*. For example:

```
?- mao.d2r(95,:v)
-> ( 1.658063 ) := 1.00 (0.000) 1
?- mao.d2r(:v,1.658)
-> ( 94.996402 ) := 1.00 (0.000) 1
```

mao.exp

`mao.exp(`*number*`|`*variable*`,`*number*`|`*variable*`)`

This *primitive* will unify or bind the second *term* with *e* raised to the power of the first *term*. For example:

```
?- mao.exp(2,:v)
-> ( 0.301030 ) := 1.00 (0.000) 1
```

If the second *term* is a *number* and the first one is an unbound *variable*, the *primitive* will bind the *variable* with the inverse operation:

```
?- mao.exp(:v,0.301030)
-> ( 2.000000 ) := 1.00 (0.000) 1
```

mao.floor

`mao.floor(`*number*`|`*variable*`,`*number*`|`*variable*`)`

This *primitive* will unify or bind the second *term* with the largest integer value less than or equal to the first *term*. For example:

```
?- mao.floor(2.145,:x)
-> ( 2 ) := 1.00 (0.000) 1
?- mao.floor(2.145,2)
-> (  ) := 1.00 (0.000) 1
?- mao.floor(6,:x)
-> ( 6 ) := 1.00 (0.000) 1
```

If the second *term* is a *number* and the first one is an unbound *variable*, the *primitive* will bind the *variable* with a *range* value:

```
?- mao.floor(:r,4)
-> ( <4|4.999999> ) := 1.00 (0.000) 1
```

mao.log

`mao.log(`*number*`|`*variable*`,`*number*`|`*variable*`)`

This *primitive* will unify or bind the second *term* with the natural logarithm (base-e logarithm) of the first *term*. For example:

```
?- mao.log(2.7,:x)
-> ( 0.993252 ) := 1.00 (0.000) 1
```

If the second *term* is a *number* and the first one is an unbound *variable*, the *primitive* will bind the *variable* with the inverse operation:

```
?- mao.log(:v,0.993252)
-> ( 2.700001 ) := 1.00 (0.000) 1
```

mao.log10

`mao.log10(`*number*`|`*variable*`,`*number*`|`*variable*`)`

This *primitive* will unify or bind the second *term* with the common logarithm (base-10 logarithm) of the first *term*. For example:

```
?- mao.log10(31.62,:v)
-> ( 1.499962 ) := 1.00 (0.000) 1
```

If the second *term* is a *number* and the first one is an unbound *variable*, the *primitive* will bind the *variable* with the inverse operation:

```
?- mao.log10(:v,1.5)
-> ( 31.622777 ) := 1.00 (0.000) 1
```

mao.modf

`mao.modf(`*number*`|`*variable*`,`*number*`|`*variable*`,`*number*`|`*variable*`)`

This *primitive* will unify or bind the second and third *terms* with the integer and fractional parts the first *term*. For example:

```
?- mao.modf(3.14,:i,:f)
-> ( 3 , 0.140000 ) := 1.00 (0.000) 1
?- mao.modf(3.14,:i,0.14)
-> ( 3 ) := 1.00 (0.000) 1
?- mao.modf(3.14,3,:f)
-> ( 0.140000 ) := 1.00 (0.000) 1
```

If the second and third *terms* is a *number* and the first one is an unbound *variable*, the *primitive* will bind the *variable* with a floating pont value created from the integer and fractional values:

```
?- mao.modf(:v,3,0.14)
-> ( 3.140000 ) := 1.00 (0.000) 1
```

mao.pow

`mao.pow(`*number*|*variable*,*number*|*variable*,*number*|*variable*`)`

The `mao.pow` *primitive* will unify or bind its third *terms* with the value of its first *term* raised to the power of its second *term*. For example:

```
?- mao.pow(8,3,:v)
-> ( 512 ) := 1.00 (0.001) 1
```

If the first or second *terms* are variables (but not at the same time), the *primitive* will bind them to the corresponding value which will make the operation work (invser power). For example:

```
?- mao.pow(8,:p,512)
-> ( 3 ) := 1.00 (0.000) 1
?- mao.pow(:v,3,512)
-> ( 8.000000 ) := 1.00 (0.001) 1
```

mao.round

`mao.round(`*number*|*variable*,*number*|*variable*`)`

This *primitive* will unify or bind the second *term* with the nearest integer value to the first *term*. For example:

```
?- mao.round(2.1,:v)
-> ( 2 ) := 1.00 (0.000) 1
?- mao.round(2.5,:v)
-> ( 3 ) := 1.00 (0.000) 1
?- mao.round(2.9,:v)
-> ( 3 ) := 1.00 (0.000) 1
```

If the second *term* is a *number* and the first one is an unbound *variable*, the *primitive* will bind the *variable* with a *range* value:

```
?- mao.round(:r,3)
-> ( <2.500001|3> ) := 1.00 (0.000) 1
```

mao.sign

`mao.sign(`*number*,*number*|*variable*`)`

This *primitive* will unify or bind the second *term* with the sign of the first *term*. For example:

```
?- mao.sign(42,:s)
-> ( 1 ) := 1.00 (0.000) 1
?- mao.sign(-42,:s)
-> ( -1 ) := 1.00 (0.000) 1
```

mao.sin

`mao.sin(`*number*`,`*number*`|`*variable*`)`

The *primitive* `mao.sin` will unify or bind its second *term* with the sine of the angle value (in degrees) given as first *term*. If the first *term* is an unbound *variable* and the second is a *number* then the *primitive* will unify the first *term* with the arc-sine. For example:

```
?- mao.sin(30,:v)
-> ( 0.500000 ) := 1.00 (0.000) 1
?- mao.sin(:a,0.5)
-> ( 30.000000 ) := 1.00 (0.000) 1
```

mao.sqrt

`mao.sqrt(`*number*`|`*variable*`,`*number*`|`*variable*`)`

This *primitive* will unify or bind its second *terms* with the square root of its first *term*. For example:

```
?- mao.sqrt(25,:v)
-> ( 5 ) := 1.00 (0.001) 1
```

If the first *term* is an unbound *variable* and the second *term* is a number, the inverse square root will computed:

```
?- mao.sqrt(:v,5)
-> ( 25 ) := 1.00 (0.000) 1
```

5.11 Miscellaneous

Hard to group *primitives* are contained in this category.

fzz.exists

`fzz.exists(`*symbol*`)`

This *primitive* will resolve with a *truth value* of `1` if its sole *term* is the label of a existing *elemental*, otherwise `0`.

fzz.labels

`fzz.labels(`*variable*`)`

This *primitive* will unify its sole *term* with a *list* containing the labels of all the *elemental* objects on the *substrate*. For example:

```
?- fzz.labels(:l)
-> ( [fzz.collect, fzz.eval, fzz.evently] ) := 1.00 (0.000) 1
```

fzz.lst

`fzz.lst(`*variable*`|`*list*`)`
`fzz.lst(`*symbol*`,`*variable*`|`*list*`)`

This *primitive* will unify it's last *term* with a list containing the GUID (as *guid term*) of all the *elemental* objects on the substrate. When two *terms* are provided, the first one is expected to be a *symbol*, indicating which group of objects to be listed. Calling this *primitive* will only work when offloaded. For example:

```
?- &fzz.lst(:l)
-> ( ['guid("263e7d79-c5e4-1f48-6a99-c8c022a2dbf3"), 'guid("1e9618ff-8e93-0147-efb6-5527b88c99cb"),
     'guid("8cbb3f79-6456-d94b-3393-8766fb3d4c72"), 'guid("f4608c21-6a8f-ab4c-4d92-5b092fa4171e"),
    'guid("40b3f684-f545-0241-99be-998167b99ab6"), 'guid("a099afdb-93a7-db4c-40b8-0341ea987ed9"), '
    guid("71cfade6-3cab-c34e-3ca6-e7a43e6fb5f7"), 'guid("330a4f04-e64c-8949-ec9e-83490c365dcb"), '
    guid("2aeb490e-bc61-2e43-99ac-3e6a11834049"), 'guid("2b5a6fef-7b4b-394b-dfa5-261d8c07a6f6"), '
    guid("2541b553-b2bc-444b-e2ac-398a9e75229c"), 'guid("7d6cb4d0-1012-554f-efbb-21e68971e496"), '
    guid("2148bc22-1f6c-0443-c59a-ed6185699715"), 'guid("24da8140-5f87-9849-be9a-f7c4aa4d5c0c"), '
    guid("91448fe7-0dd7-bc43-2795-3ec6e3a71537"), 'guid("9c5ab238-d08e-d148-a998-203f8096878c"), '
    guid("adae7700-7605-114d-2492-45e3d69b9a23"), 'guid("29a0e5fe-a979-7542-0bb3-c275d09e0190"), '
    guid("0513542a-ed1b-474b-e099-fa92fa234295"), 'guid("e291a61e-fb51-0747-9da4-d03bf24d22a4")] )
    := 1.00 (0.002) 1
```

fzz.parse

`gid.parse(`*string*`|`*variable*`)`

This *primitive* will parse a *string* given as first term and unify or substitue its parsing into a *fizz term* with the second *term*. If the string cannot be parsed into a valid *term*, the call will resolve with a *truth value* of `0`. Here's an example:

```
?- fzz.parse("[a,b,c(5),d]",:v)
-> ( [a, b, c(5), d] ) := 1.00 (0.001) 1
```

fzz.stats

`gid.stats(`*frame*`|`*variable*`)`

This *primitive* will unify or substitue its only *term* with a *frame* containing statistics about the *runtime environment*. Note that it can only be called when offloaded. Here's an example:

```
?- &fzz.stats(:f)
-> ( {elementals = 7, knowledges = 1, statements = 0, prototypes = 1, uptime = 7.601939, lag = 0,
    queries = 0, replies = 0, squibs = 0, mpeak = 112, msize = 112, stime = 0, utime = 0} ) := 1.00
     (0.001) 1
```

gid.make

`gid.make(`*guid*`|`*variable*`)`

This *primitive* will unify or substitue its only *term* with a randomly generated *guid term*. Here's an example:

```
?- gid.make(:g)
-> ( 'guid("e30f998a-020d-fd4c-c0b8-e384d2dc8020") ) := 1.00 (0.001) 1
?- gid.make(:g)
-> ( 'guid("ce0c25e6-5adc-9e48-0c80-57b70db9a2e0") ) := 1.00 (0.000) 1
```

gid.sym

`gid.sym(`*symbol*|*variable*`)`

This *primitive* will unify or substitue its *term* with a randomly generated *symbol*. Here's an example:

```
?- gid.sym(:g)
-> ( yzrxzqubtaxcqrubbuyeaaqfcuysbfuw ) := 1.00 (0.000) 1
```

The generated symbol is a *globally unique identifier* (GUID).

gid.str

`gid.str(`*symbol*|*variable*`)`

This *primitive* will unify or substitue its *term* with a randomly generated *string*. Here's an example:

```
?- gid.str(:g)
-> ( "005a7ce9-433f-574c-d1ba-5a03240eb98e" ) := 1.00 (0.000) 1
```

var.capture

`var.capture(`*variable*, *list?*`)`

This *primitive* will unify or substitue its first *term* with a *frame* containing all bound variables and their values. If a second *term* is provided, it is expected to be the name of all the *variables* which are not to be included in the capture. Here's an example:

```
?- set(:A,1), set(:B,2), set(:C,3), var.capture(:f)
-> ( {A = 1, B = 2, C = 3} ) := 1.00 (0.000) 1
?- set(:A,1), set(:B,2), set(:C,3), var.capture(:f,[C])
-> ( {A = 1, B = 2} ) := 1.00 (0.000) 1
```

var.collect

`var.collect(`*symbols+*, *variable*`)`

This *primitive* will unify or substitue its last *term* with a list containing the values of all the bound *variables* which label was provided as *terms* to the *primitive*. For example:

```
?- set(:A,1), set(:B,2), set(:C,3), var.collect(A,C,:f)
-> ( [1, 3] ) := 1.00 (0.000) 1
```

var.defu

`var.defu(`*term*, *variable*`)`

This *primitive* performs the inverse operation of the primitive `var.tofu`. For example

```
    ?- var.tofu([hello,:x,how(are(:y?{friend=yes}))],:L), var.defu(:L,:f)
    -> ( :x , :y ? {friend = yes} , [hello, :x, how(are(:y ? {friend = yes}))] ) := 1.00 (0.000) 1
```

var.make

`var.make(`*number*, *variable*`)`

This *primitive* will unify or substitue its second *term* with a *list* containing as many (randomly named) unbound *variables* as requested with the first *term*. Here's an example:

```
?- var.make(3,:l)
-> ( [:bqtkd, :juwcm, :fbdpn] ) := 1.00 (0.000) 1
```

var.release

`var.release(`*frame*`)`

This *primitive* will thake the content of the frame given as its only *term* and bind a *variable* for each of the label/value pairs. For example:

```
?- var.release({a = 1, b = 2}), console.puts(:a," ",:b)
1 2
-> ( 1 , 2 ) := 1.00 (0.000) 1
```

var.tofu

`var.tofu(`*term*, *variable*`)`

This *primitive* will unify or substitue its second *term* with a copy of its first *term* where all unbound *variables* will be replaced by a special *functor* whose label will always be `var` and with the label of the *variable* as first *term*. If the *variable* has a constraint, the *functor* will have an arity of two. Its second *term* will be the constraint. For a *wildcard variable*, the first *term* will be the *symbol* `wildcard`. Here's an example:

```
?- var.tofu([hello,:x,how(are(:y?{friend=yes}))],:l)
-> ( :x , :y ? {friend = yes} , [hello, var(x), how(are(var(y, {friend = yes})))] ) := 1.00 (0.000)
     1
```

To invert the action of the primitive, use the primitive `var.defu`.

5.12 Quirk

All *primitives* related to handling *quirks* are grouped in this category.

qrk.head

`qrk.head(`*quirk*, *term*`)`

This *primitive* will unify or substitue its second *term* with the head (the first element)) in the *quirk* passed as first *term*:

```
?- qrk.head(hello^5,:h)
-> ( hello ) := 1.00 (0.000) 1
```

If the *term* is not a *quirk*, the *term* will be unified with the second *term*.

qrk.make

`qrk.make(`*term*`,`*term*`,`*quirk*`|`*variable*`)`

This *primitive* will unify or substitue its third *term* with a *quirk* build from its first and second *term*:

```
?- qrk.make(hello,5,:q)
-> ( hello^5 ) := 1.00 (0.000) 1
```

qrk.tail

`qrk.tail(`*quirk*`,`*term*`)`

This *primitive* will unify or substitue its second *term* with the tail (the second element)) in the *quirk* passed as first *term*:

```
?- qrk.head(hello^5,:h)
-> ( 5 ) := 1.00 (0.000) 1
```

If the *term* is not a *quirk*, the *term* will be unified with the second *term*.

5.13 Random

This section describes *primitives* that generate random *numbers*.

rnd.list

`rnd.list(`*number*`,`*list*`,`*term*`|`*variable*`)`

This *primitive* will unify or bind the third *term* with a series of randomly picked *terms* from the *list* content given as second *term*. The first *term* is the count of random elements to be provided. For example:

```
?- rnd.list(2,[1,2,3,4,5,6,7,8,9,10],:v)
-> ( 4 ) := 1.00 (0.000) 1
-> ( 6 ) := 1.00 (0.001) 2
```

rnd.real

`rnd.real(`*number*`,`*number*`|`*variable*`,`*number?*`,`*number?*`)`

This *primitive* will unify or bind the second *term* with a series of (floating point) random *number* picked in the range defined in between the third and fourth *terms*. The first *term* is the count of random *numbers* to be provided. For example:

```
?- rnd.real(5,:v,1,100)
-> ( 86.598612 ) := 1.00 (0.000) 1
-> ( 80.759627 ) := 1.00 (0.000) 2
-> ( 41.959139 ) := 1.00 (0.000) 3
-> ( 30.452654 ) := 1.00 (0.001) 4
-> ( 20.528407 ) := 1.00 (0.001) 5
```

When no range is provided, the random number will all be in between `0` and `1`:

```
?- rnd.real(5,:v)
-> ( 0.791721 ) := 1.00 (0.000) 1
-> ( 0.829935 ) := 1.00 (0.000) 2
-> ( 0.496939 ) := 1.00 (0.000) 3
-> ( 0.007982 ) := 1.00 (0.001) 4
-> ( 0.891288 ) := 1.00 (0.001) 5
```

rnd.rsnd

`rnd.rsnd(`*number*`,`*number*`,|`*variable*`,`*number*`,`*number*`)`

This *primitive* will unify or bind the third *term* with a serie of (floating point) random *numbers* picked from a standard normal deviation where the first *term* is the *mean* and the second is the *standard deviation.* The first *term* is the count of random *numbers* to be provided. For example:

```
?- rnd.rsnd(10,:x,0,1)
-> ( -1 ) := 1.00 (0.001) 1
-> ( 0.488077 ) := 1.00 (0.001) 2
-> ( -2 ) := 1.00 (0.002) 3
-> ( 0 ) := 1.00 (0.002) 4
-> ( 0.807786 ) := 1.00 (0.002) 5
-> ( 0.913344 ) := 1.00 (0.002) 6
-> ( 0 ) := 1.00 (0.003) 7
-> ( 0.327671 ) := 1.00 (0.003) 8
-> ( 0.000954 ) := 1.00 (0.003) 9
-> ( 0.762686 ) := 1.00 (0.004) 10
```

rnd.uint

`rnd.uint(`*number*`,`*number*`|`*variable*`,`*number?*`,`*number?*`)`

This *primitive* will unify or bind the second *term* with a series of (unsigned integer) random *numbers* picked in the range defined between the third and fourth *terms*. The first *term* is the count of random *numbers* to be provided. For example:

```
?- rnd.uint(5,:v,1,100)
-> ( 36 ) := 1.00 (0.000) 1
-> ( 44 ) := 1.00 (0.000) 2
-> ( 90 ) := 1.00 (0.001) 3
-> ( 17 ) := 1.00 (0.001) 4
-> ( 55 ) := 1.00 (0.001) 5
```

When no range is provided, the random *numbers* will all be in between `0` and the maximum value for a 64-bit unsigned integer:

```
?- rnd.uint(5,:v)
-> ( 227958570 ) := 1.00 (0.000) 1
-> ( 2008933850 ) := 1.00 (0.000) 2
-> ( 834617219 ) := 1.00 (0.001) 3
-> ( 351245525 ) := 1.00 (0.001) 4
-> ( 1962305856 ) := 1.00 (0.001) 5
```

rnd.sint

`rnd.sint(`*number*`,`*number*`|`*variable*`,`*number?*`,`*number?*`)`

This *primitive* will unify or bind the second *term* with a series of (signed integer) random *numbers* picked in the range defined between the third and fourth *terms*. The first *term* is the count of random *numbers* to be provided. For example:

```
?- rnd.sint(3,:v,-100,100)
-> ( -48 ) := 1.00 (0.001) 1
-> ( 90 ) := 1.00 (0.001) 2
-> ( -29 ) := 1.00 (0.001) 3
```

When no range is provided, the random *numbers* will all be in between the possible value for a 64-bit signed integer:

```
?- rnd.sint(5,:v)
-> ( -3832553529235211065 ) := 1.00 (0.001) 1
-> ( 2840651865658580059 ) := 1.00 (0.001) 2
-> ( -4585361323621985541 ) := 1.00 (0.001) 3
-> ( 8886134878488290534 ) := 1.00 (0.001) 4
-> ( 4799459735435763595 ) := 1.00 (0.001) 5
```

5.14 Range

This section describes *primitives* that handle *ranges* or generate *numbers* based on range.

rng.clamp

`rng.clamp(`*range*`,`*number*`,`*number*`|`*variable*`)`

The *primitive* will unify or bind its third *term* with its second *term* clamped to the *range* provided as first *term*. For example:

```
?- rng.clamp(<1|10>,11,:v)
-> ( 10 ) := 1.00 (0.001) 1
?- rng.clamp(<1|10>,-2,:v)
-> ( 1 ) := 1.00 (0.001) 1
?- rng.clamp(<1|10>,5,:v)
-> ( 5 ) := 1.00 (0.001) 1
```

rng.inter

`rng.inter(`*range*`,`*range*`,`*range*`|`*variable*`)`

This *primitive* unifies/binds its third *term* with the intersection of the two *ranges* provided as the first *terms*. For example:

```
?- rng.inter(<10.3|26.7>,<17.34|43>,:r)
-> ( <17.340000|26.700000> ) := 1.00 (0.000) 1
```

If there is no intersection between the two *ranges*, the call will resolve with a *truth value* of `0`.

rng.max

`rng.max(`*range*`,`*number*`|`*variable*`)`

The `rng.max` *primitive* will unify or bind its second *term* with the maximum value of the *range* given as first *term*. For example:

```
?- rng.max(<10.3|26.7>,:max)
-> ( 26.700000 ) := 1.00 (0.000) 1
```

rng.min

`rng.min(`*range*`,`*number*`|`*variable*`)`

The `rng.min` *primitive* will unify or bind its second *term* with the minimum value of the *range* given as first *term*. For example:

```
?- rng.min(<10.3|26.7>,:min)
-> ( 10.300000 ) := 1.00 (0.000) 1
```

rng.norm

`rng.norm(`*range*`,`*number*`,`*number*`|`*variable*`)`

This *primitive* will unify or bind its third *term* with the normalized value of the second *term*. For example:

```
?- rng.norm(<0|10>,2,:v)
-> ( 0.200000 ) := 1.00 (0.000) 1
?- rng.norm(<0|10>,11,:v)
-> ( 1 ) := 1.00 (0.000) 1
?- rng.norm(<0|10>,3.85,:v)
-> ( 0.385000 ) := 1.00 (0.000) 1
```

rng.not

`rng.not(`*range*`,`*number*`)`

The `rng.not` *primitive* will resolve to a *truth value* of `0` if the second *term* is a *number* whose value is within the *range* given as first *term*. For example:

```
?- rng.not(<0|10>,3.85)
-> (  ) := 0.00 (0.000) 1
?- rng.not(<0|10>,11)
-> (  ) := 1.00 (0.000) 1
```

rng.inc

`rng.inc(`*range*`,`*number*`)`

The `rng.inc` *primitive* will resolve to a *truth value* of `1.0` if the second *term* is a *number* whose value is within the *range* given as first *term*. For example:

```
?- rng.inc(<10.3|26.7>,11)
-> (  ) := 1.00 (0.000) 1
?- rng.inc(<10.3|26.7>,10)
-> (  ) := 0.00 (0.000) 1
```

Unlike `rng.span`, this *primitive* will not generate values within the range if the second *term* is an unbound *variable*.

rng.span

`rng.span(`*range*`,`*number*`,`*number*`|`*variable*`)`

The *primitive* will unify or bind its third *term* with any *number* that is included in the *range* provided as first *term*. The second *term* is the difference between consecutive values to be used to traverse the range. For example:

```
?- rng.span(<0|1>,0.1,:v)
-> ( 0 ) := 1.00 (0.001) 1
-> ( 0.100000 ) := 1.00 (0.002) 2
-> ( 0.200000 ) := 1.00 (0.002) 3
-> ( 0.300000 ) := 1.00 (0.003) 4
-> ( 0.400000 ) := 1.00 (0.003) 5
-> ( 0.500000 ) := 1.00 (0.004) 6
-> ( 0.600000 ) := 1.00 (0.004) 7
-> ( 0.700000 ) := 1.00 (0.005) 8
-> ( 0.800000 ) := 1.00 (0.005) 9
-> ( 0.900000 ) := 1.00 (0.006) 10
-> ( 1 ) := 1.00 (0.006) 11
```

rng.union

`rng.union(`*range*`,`*range*`,`*range*`|`*variable*`)`

This *primitive* unifies/binds its third *term* with the union of the two *ranges* provided as the first *terms*. For example:

```
?- rng.union(<10.3|26.7>,<17.34|43>,:r)
-> ( <10.300000|43> ) := 1.00 (0.000) 1
```

rng.uint

`rng.uint(`*number*`,`*number*`,`*number*`|`*variable*`)`

This *primitive* will unify or bind its third *term* with any *unsigned number* between the first and second *terms*. For example:

```
?- rng.uint(1,10,11)
-> (  ) := 0.00 (0.001) 1
?- rng.uint(1,10,2)
-> (  ) := 1.00 (0.000) 1
```

If the third *term* is an unbound variable, the *primitive* will generate as many solutions as there are unsigned integers in the range:

```
?- rng.uint(1,10,:x)
-> ( 1 ) := 1.00 (0.001) 1
-> ( 2 ) := 1.00 (0.001) 2
-> ( 3 ) := 1.00 (0.001) 3
-> ( 4 ) := 1.00 (0.001) 4
-> ( 5 ) := 1.00 (0.001) 5
-> ( 6 ) := 1.00 (0.002) 6
-> ( 7 ) := 1.00 (0.002) 7
-> ( 8 ) := 1.00 (0.002) 8
-> ( 9 ) := 1.00 (0.002) 9
-> ( 10 ) := 1.00 (0.002) 10
```

rng.rand

`rng.rand`(*range*, *number*| *variable*)

This *primitive* will unify or bind its second *term* with a random *number* picked from the first *term*. For example:

```
?- rng.rand(<0|1>,:v)
-> ( 0.359032 ) := 1.00 (0.001) 1
?- rng.rand(<0|1>,:v)
-> ( 0.751194 ) := 1.00 (0.000) 1
?- rng.rand(<0|1>,:v)
-> ( 0.320658 ) := 1.00 (0.000) 1
```

rng.real

`rng.real`(*number*, *number*, *number*| *variable*)

This *primitive* will unify or bind its third *term* with any *real number* between the first and second *terms*. For example:

```
?- rng.real(1,10,11)
-> (  ) := 0.00 (0.001) 1
?- rng.real(1,10,2)
-> (  ) := 1.00 (0.000) 1
```

If the third *term* is an unbound variable, the *primitive* will generate as many solutions as there are unsigned integers in the range:

```
?- rng.real(1,10,:x)
-> ( 1 ) := 1.00 (0.001) 1
-> ( 2 ) := 1.00 (0.001) 2
-> ( 3 ) := 1.00 (0.001) 3
-> ( 4 ) := 1.00 (0.001) 4
-> ( 5 ) := 1.00 (0.001) 5
-> ( 6 ) := 1.00 (0.002) 6
-> ( 7 ) := 1.00 (0.002) 7
-> ( 8 ) := 1.00 (0.002) 8
-> ( 9 ) := 1.00 (0.002) 9
-> ( 10 ) := 1.00 (0.002) 10
```

5.15 Regexp

This section describes *primitives* that handle regular expressions.

rex.make

`rex.make(`*string*`,`*regexp*`|`*variable*`)`
`rex.make(`*string*`,`*list*`,`*regexp*`|`*variable*`)`

This *primitive* creates a new *regexp* using the pattern provided as the first *term* and an optional *list* of flags, and unify it with the last *term*. For example:

```
?- rex.make("(the|a)?\s?(dog|cat)\sis\s(wet|cold|sick)",[caseless],:r), rex.match(:r,"dog is wet",:
    l)
-> ( 'regexp("(the|a)?\s?(dog|cat)\sis\s(wet|cold|sick)",[CASELESS]) , ["dog is wet", "", "dog", "
    wet"] ) := 1.00 (0.000) 1
```

For the list of supported compilation flags, see Section **??** on page ??.

rex.match

`rex.match(`*regexp*`,`*string*`,`*list*`|`*variable?*`)`

The *primitive* `rex.match` will match a *string* given as its second *term* with the *regular expression* provided as first *term* and will resolve to a *truth value* of `1.0` if it is a match.

```
?- rex.match('regexp("[a|b]+"),"ABAB")
-> (  ) := 0.00 (0.000) 1
?- rex.match('regexp("^[a|b]+$"),"abab")
-> (  ) := 1.00 (0.000) 1
?- rex.match('regexp("^[a|b]+$"),"ababc")
-> (  ) := 0.00 (0.000) 1
```

If a third *term* is provided, the *primitive* will unify it with all the matchs between the *regexp* and the *string*:

```
?- rex.match('regexp("\d+"),"12 drummers drumming, 11 pipers piping, 10 lords a-leaping",:l)
-> ( ["12", "11", "10"] ) := 1.00 (0.000) 1
```

5.16 Symbol

This section describes *primitives* related to handling *symbols*.

sym.cat

`sym.cat(`*term+*`,`*string*`|`*variable*`)`

This *primitive* will unify or substitue the concatenation of all its *terms* but the last one, with the last one. Then turns that into a *symbol*. For example:

```
?- sym.cat(hello,".",4,:x)
-> ( hello.4 ) := 1.00 (0.001) 1
```

If a *list* is used as a *term*, the *terms* it contains will be used as if they had been passed to the *primitive* itself. For example:

```
?- sym.cat([a,5,c],:s)
-> ( a_5_c ) := 1.00 (0.000) 1
```

sym.cmp

`sym.cmp(`*symbol*`,`*symbol*`,`*number*`|`*variable*`,`*symbol?*`)`

The `sym.cmp` *primitive* will unify or substitue its third *term* with the result of the comparison of its first two *symbol terms*. When the first *term* is greater than the second *term*, the third *term* will unify with the value `1`. If less, it will be unified with the value `-1`. When both *strings* are identical, the value will be `0`. For example:

```
?- sym.cmp(hello,hello4,:c)
-> ( -1 ) := 1.00 (0.001) 1
?- sym.cmp(hello,hello,:c)
-> ( 0 ) := 1.00 (0.000) 1
?- sym.cmp(hello,Hello,:c)
-> ( 1 ) := 1.00 (0.000) 1
?- sym.cmp(hello,Hello,:c,insensitive)
-> ( 0 ) := 1.00 (0.000) 1
```

The optional fourth *term* can be the *symbol* `insensitive` to indicate that the comparison must be case insensitive.

sym.cut

`sym.cut(`*symbol*`,`*symbol*`|`*list*`,`*symbol*`|`*variable*`)`

The `sym.cut` *primitive* will unify or substitue its third *term* with its first *term* where any matching ending has been removed. The second *term* can either be a list of potential ending *symbols* or a *symbol*. For example:

```
?- sym.cut(hello,lo,:s)
-> ( hel ) := 1.00 (0.001) 1
?- sym.cut(hello,la,:s)
-> ( hello ) := 1.00 (0.000) 1
?- sym.cut(hello,[la,lo],:s)
-> ( hel ) := 1.00 (0.000) 1
?- sym.cut(hella,[la,lo],:s)
-> ( hel ) := 1.00 (0.000) 1
```

sym.end

`sym.end(`*symbol*`,`*symbol*`|`*list*`)`

The `sym.end` *primitive* will resolve to a *truth value* of `1` if the second *term* is found at the end of the first *term*. If the second *term* is a list, any of the *symbols* it may contains will be tested. For example:

```
?- sym.end(hello,lo)
-> (  ) := 1.00 (0.000) 1
?- sym.end(hello,la)
-> (  ) := 0.00 (0.000) 1
?- sym.end(hello,[lo,la])
```

```
-> (  ) := 1.00 (0.000) 1
?- sym.end(hella,[lo,la])
-> (  ) := 1.00 (0.000) 1
?- sym.end(hellu,[lo,la])
-> (  ) := 0.00 (0.000) 1
```

sym.stem

`sym.stem(`*symbol*, *symbol*| *variable*`)`

The `sym.stem` *primitive* will unify or substitue its second *term* with a stemmed version of first *term* using Martin Porter's Stemming algorithm. For example:

```
?- sym.stem(cats,:s)
-> ( cat ) := 1.00 (0.000) 1
```

sym.sub

`sym.sub(`*symbol*, *number*, *number*, *symbol*| *variable*`)`

The `sym.sub` *primitive* will unify or substitue its fourth *terms* with a subpart of the *symbol* given as first *term*. The subpart is defined by an offset (second *term*) and a length (third *term*). For example:

```
?- sym.sub(truck,0,1,:c)
-> ( t ) := 1.00 (0.001) 1
```

sym.tokenize

`sym.tokenize(`*symbol*, *string*, *list*| *variable*, *list?*`)`

This *primitive* will unify or substitue its third *term* with a *list* of tokens, which are substring of its first *term* separated by any of the characters that are part of the second *term*. For example:

```
?- sym.tokenize(a.b.c.d,".",:l)
-> ( [a, b, c, d] ) := 1.00 (0.001) 1
```

If the first *term* is an unbound *variable* and the 3rd *term* is a *list*, the *primitive* will generate a symbol from the concatenation of all items in the list (but only if the *terms* are *string, symbol, number* or *list*). For example:

```
?- sym.tokenize(:s,".",[a,b,c,"h 4"])
-> ( a.b.c.h_4 ) := 1.00 (0.001) 1
```

sym.tolower

`sym.tolower(`*symbol*, *symbol*| *variable*`)`

The `sym.tolower` *primitive* will unify or substitue its second *term* with a copy of first *term* where all alphabetic characters have been converted to lowercase:

```
?- sym.tolower(HeLLo,:s)
-> ( hello ) := 1.00 (0.000) 1
```

sym.toupper

`sym.toupper(`*symbol*`,`*symbol*`|`*variable*`)`

The `sym.toupper` *primitive* will unify or substitue its second *term* with a copy of first *term* where all alphabetic characters have been converted to uppercase:

```
?- sym.symbol(HeLLo,:s)
-> ( HELLO ) := 1.00 (0.000) 1
```

5.17 String

This section describes *primitives* related to handling *strings*.

str.cat

`str.cat(`*term+*`,`*string*`|`*variable*`)`

This *primitive* will unify or substitue the concatenation of all its *terms* but the last one, with the last one. For example:

```
?- str.cat(hello," ",how," ",are," ",you,:s)
-> ( "hello how are you" ) := 1.00 (0.000) 1
```

str.cmp

`str.cmp(`*string*`,`*string*`,`*number*`|`*variable*`,`*symbol?*`)`

The `str.cmp` *primitive* will unify or substitue its third *term* with the result of the comparison of its first two *string terms*. When the first *term* is greater than the second *term*, the third *term* will unify with the value `1`. If less, it will be unified with the value `-1`. When both *strings* are identical, the value will be `0`. For example:

```
?- str.cmp("abcdef","ABCDEF",:c)
-> ( 1 ) := 1.00 (0.000) 1
?- str.cmp("abcdef","ABCDEF",:c,insensitive)
-> ( 0 ) := 1.00 (0.001) 1
```

The optional fourth *term* can be the *symbol* `insensitive` to indicate that the comparison must be case insensitive.

str.dist

`str.dist(`*string*`,`*string*`,`*number*`|`*variable*`)`

This *primitive* will unify or substitue its third *term* with the Levenshtein distance between the two strings given as first *terms*. For example:

```
?- str.dist("Hello world","Hello World",:d)
-> ( 1 ) := 1.00 (0.000) 1
?- str.dist("Truck","car",:d)
-> ( 5 ) := 1.00 (0.000) 1
```

str.end

`str.end(`*string*`,`*string*`|`*list*`)`

The `str.end` *primitive* will resolve to a *truth value* of `1` if the second *term* is found at the end of the first *term*. If the second *term* is a list, any of the *strings* it may contains will be tested. For example:

```
?- str.end("hello","lo")
-> (  ) := 1.00 (0.000) 1
?- str.end("hello","la")
-> (  ) := 0.00 (0.000) 1
?- str.end("hello",["la","lo"])
-> (  ) := 1.00 (0.000) 1
?- str.end("hellu",["la","lo"])
-> (  ) := 0.00 (0.000) 1
```

str.find

`str.find(`*string*`,`*string*`,`*number?*`|`*variable?*`)`

The `str.find` *primitive* will unify or substitue its third *term* with the offset (starting from 0) within its first *term* where the second *term* was find. If there is no occurence of the second *term*, the third will unify with the value `-1`. For example:

```
?- str.find("abcdef","bc",:o)
-> ( 1 ) := 1.00 (0.000) 1
?- str.find("abcdef","ef",:o)
-> ( 4 ) := 1.00 (0.000) 1
?- str.find("abcdef","ef",4)
-> (  ) := 1.00 (0.000) 1
?- str.find("abcdef","g",:p)
-> ( -1 ) := 1.00 (0.000) 1
```

The *primitive* will generate as many solutions as there is occurences of the second *term* in the *string*:

```
?- str.find("abcdefcc","c",:p)
-> ( 2 ) := 1.00 (0.000) 1
-> ( 6 ) := 1.00 (0.001) 2
-> ( 7 ) := 1.00 (0.001) 3
```

If no third *term* is given, then the *primitive* will resolve to a *truth value* of `1` if the second *term* is found anywhere in the first *term*.

str.flip

`str.flip(`*string*`,`*string*`|`*variable*`)`

The `str.flip` *primitive* will unify or substitue its second *term* with a *string* containing the content of the first *term* inverted:

```
?- str.flip("hello, world!",:s)
-> ( "!dlrow ,olleh" ) := 1.00 (0.001) 1
```

str.head

`str.head(`*string*`,`*string*`)`
`str.head(`*string*`,`*string*`,`*symbol*`)`

The *primitive* will resolve to a *truth value* of `1` if the *string* given as second *term* is the start of the *string* given as first *term*, `0` otherwise. For example:

```
?- str.head("hello world!","hello")
-> (  ) := 1.00 (0.000) 1
?- str.head("hello world!","world")
-> (  ) := 0.00 (0.000) 1
?- str.head("hello world!","HELLO")
-> (  ) := 0.00 (0.000) 1
```

An optional third *term* (a *symbol*) can indicate of the case of the strings should not matter (`insensitive`) or should (`sensitive`) in the *comparison*. When no third *term* is specified, the default behavior is to be case sensitive:

```
?- str.head("hello world!","HELLO",insensitive)
-> (  ) := 1.00 (0.000) 1
```

str.length

`str.length(`*string*`,`*number*`|`*variable*`)`

This *primitive* will unify or substitue its second *term* with the length (that is the number of characters) in the *string* passed as first *term*.

```
?- str.length("hello, world!",:l)
-> ( 13 ) := 1.00 (0.000) 1
```

str.rest

`str.rest(`*string*`,`*number*`,`*string*`|`*variable*`)`

The `str.rest` *primitive* will unify or substitue its third *terms* with a subpart of the *string* given as first *term*. The subpart is defined as starting at a given position (the second *term*) in the *string* and runs up to the end of the *string*. For example:

```
?- str.rest("hello, how are you?",7,:w)
-> ( "how are you?" ) := 1.00 (0.001) 1
```

str.sub

`str.sub(`*string*`,`*number*`,`*number*`,`*string*`|`*variable*`)`

The `str.sub` *primitive* will unify or substitue its fourth *terms* with a subpart of the *string* given as first *term*. The subpart is defined by an offset (second *term*) and a length (third *term*). For example:

```
?- str.sub("hello, how are you?",7,3,:w)
-> ( "how" ) := 1.00 (0.000) 1
```

str.swap

`str.swap(`*string*`,`*list*`,`*string*`|`*variable*`)`

The `str.swap` *primitive* will unify or substitue its third *term* with it's first *term* where all occurences of specified *strings* will have been replaced by provided *strings*. For example:

```
?- str.swap("GATTACA",[["A","T"],["C","G"],["G","C"],["T","A"]],:s)
-> ( "CTAATGT" ) := 1.00 (0.000) 1
?- str.swap("abc123abc456789abc",["abc","A"],:s)
-> ( "A123A456789A" ) := 1.00 (0.000) 1
```

str.tail

`str.tail(`*string*`,`*string*`)`
`str.tail(`*string*`,`*string*`,`*symbol*`)`

The *primitive* will resolve to a *truth value* of `1` if the *string* given as second *term* is the end of the *string* given as first *term*, `0` otherwise. For example:

```
?- str.tail("hello world!","world!")
-> (  ) := 1.00 (0.000) 1
?- str.tail("hello world!","world!")
-> (  ) := 0.00 (0.000) 1
?- str.tail("hello world!","WORLD!")
-> (  ) := 0.00 (0.000) 1
```

An optional third *term* (a *symbol*) can indicate of the case of the strings should not matter (`insensitive`) or should (`sensitive`) in the *comparison*. When no third *term* is specified, the default behavior is to be case sensitive:

```
?- str.tail("hello world!","WORLD!",insensitive)
-> (  ) := 1.00 (0.000) 1
```

str.tokenize

`str.tokenize(`*string*`,`*string*`,`*list*`|`*variable*`,`*list?*`)`

This *primitive* will unify or substitue its third *term* with a *list* of tokens, which are substring of its first *term* separated by any of the characters that are part of the second *term*. For example:

```
?- str.tokenize("66;3.14;22",";",:l)
-> ( ["66", "3.14", "22"] ) := 1.00 (0.000) 1
?- str.tokenize("66;3.14,22",";,",:l)
-> ( ["66", "3.14", "22"] ) := 1.00 (0.000) 1
```

If the first *term* is an unbound *variable* and the 3rd *term* is a *list*, the *primitive* will generate a string from the concatenation of all items in the list (but only if the *terms* are *string, symbol, number* or *list*). For example:

```
?- str.tokenize(:s," ",[a,b,c,[d,e,f]])
-> ( "a b c d e f" ) := 1.00 (0.000) 1
```

When a fourth *term* is provided, it is expected to be a *list* of *symbols* acting as *flags*. The only flag supported at the moment is `include` which instructs the *primitive* to include the delimiters as elements of the tokenized *list*. For example:

```
?- str.tokenize("66;3.14;22",";",:l,[include])
-> ( ["66", ";", "3.14", ";", "22"] ) := 1.00 (0.001) 1
```

str.tolower

`str.tolower`(*string*,*string*|*variable*)

The **`str.tolower`** *primitive* will unify or substitue its second *term* with a copy of first *term* where all alphabetic characters have been converted to lowercase:

```
?- str.tolower("HeLLo",:s)
-> ( "hello" ) := 1.00 (0.000) 1
```

str.tonum

`str.tonum`(*string*|*variable*,*number*|*variable*)

The **`str.tonum`** *primitive* will unify or substitue its second *term* with a *number* parsed from its first *term*. For example:

```
?- str.tonum("45f",:v)
-> ( 45 ) := 1.00 (0.000) 1
?- str.tonum("-125",:v)
-> ( -125 ) := 1.00 (0.000) 1
```

If the first *term* is an unbound *variable* and the second *term* is a *number*, the *primitive* will unify the *variable* with a *string* eersion of the *number*:

```
?- str.tonum(:x,12.42)
-> ( "12.420" ) := 1.00 (0.000) 1
?- str.tonum(:x,66)
-> ( "66" ) := 1.00 (0.000) 1
```

str.toupper

`str.toupper`(*string*,*string*|*variable*)

The **`str.toupper`** *primitive* will unify or substitue its second *term* with a copy of first *term* where all alphabetic characters have been converted to uppercase:

```
?- str.toupper("HeLLo",:s)
-> ( "HELLO" ) := 1.00 (0.000) 1
```

str.tosym

`str.tosym(`*string*`,`*symbol*`|`*variable*`)`

The `str.tosym` *primitive* will unify or substitue its second *term* with *symbol* based on its first *term*. For example:

```
?- str.tosym("HeLLo",:s)
-> ( HeLLo ) := 1.00 (0.000) 1
?- str.tosym("3.14",:s)
-> ( a3.14 ) := 1.00 (0.000) 1
?- str.tosym("hello, world.",:s)
-> ( hello._world. ) := 1.00 (0.000) 1
```

str.trim

`str.trim(`*string*`,`*variable*`)`
`str.trim(`*string*`,`*variable*`,`*string*`)`

This *primitive* will unify or substitue its second *term* with its first *term* trimmed of any empty spaces at the start and end of the *string*. For example:

```
?- str.trim(" this is my string  ",:s)
-> ( "this is my string" ) := 1.00 (0.001) 1
```

When a third *term* is given, it will be a *string* which content will be trimmed from the first *term* instead of the empty spaces:

```
?- str.trim("555-1234",:s,"555-")
-> ( "1234" ) := 1.00 (0.000) 1
```

str.trim.head

`str.trim.head(`*string*`,`*variable*`)`
`str.trim.head(`*string*`,`*variable*`,`*string*`)`

This *primitive* will unify or substitue its second *term* with its first *term* trimmed of any empty spaces at the start of the *string*. When a third *term* is given, it will be a *string* which content will be trimmed from the first *term*. For example:

```
?- str.trim.head(" this is my string  ",:s)
-> ( "this is my string  " ) := 1.00 (0.001) 1
?- str.trim.head("555-1234-555",:s,"555")
-> ( "-1234-555" ) := 1.00 (0.000) 1
```

str.trim.tail

`str.trim.tail(`*string*`,`*variable*`)`
`str.trim.tail(`*string*`,`*variable*`,`*string*`)`

This *primitive* will unify or substitue its second *term* with its first *term* trimmed of any empty spaces at the end of the *string*. When a third *term* is given, it will be a *string* which content will be trimmed from the first *term*. For example:

```
?- str.trim.tail(" this is my string  ",:s)
-> ( " this is my string" ) := 1.00 (0.001) 1
?- str.trim.head("555-1234-555",:s,"555")
-> ( "555-1234-" ) := 1.00 (0.000) 1
```

5.18 Typing

This section describes *primitives* that can be used to check the type of any *terms*.

is.atom

`is.atom(`*term*`)`

The *primitive* will resolve to a *truth value* of `1` if the *term* is an *atom*, `0` otherwise. For example:

```
?- is.atom(4)
-> (  ) := 1.00 (0.000) 1
?- is.atom("hello world")
-> (  ) := 1.00 (0.000) 1
?- is.atom([a,b,c,d])
-> (  ) := 0.00 (0.000) 1
?- is.atom(neat)
-> (  ) := 1.00 (0.000) 1
```

is.binary

`is.binary(`*term*`)`

The *primitive* will resolve to a *truth value* of `1` if the *term* is a *binary*, `0` otherwise. For example:

```
?- is.binary(42)
-> (  ) := 0.00 (0.000) 1
?- is.binary(hello)
-> (  ) := 0.00 (0.001) 1
?- is.binary("the quick fox ...")
-> (  ) := 0.00 (0.000) 1
?- is.binary('binary("aGVsbG8sIHdvcmxkIQA="))
-> (  ) := 1.00 (0.000) 1
```

is.bound

`is.bound(`*term*`)`

The *primitive* will resolve to a *truth value* of `1` if the *term* is a bound *variable*, `0` otherwise. For example:

```
?- is.bound(:h)
-> ( :h ) := 0.00 (0.000) 1
?- is.bound(5)
-> (  ) := 1.00 (0.000) 1
?- set(:h,5), is.bound(:h)
-> ( 5 ) := 1.00 (0.000) 1
```

When multiple *terms* are used, it will resolve to `1` if all the *terms* are bounded *variables*.

is.data

`is.data(`*term*`)`

The *primitive* will resolve to a *truth value* of `1` if the *term* is a *data*, `0` otherwise. For example:

```
?- is.data([2,4,2,0.5])
-> (  ) := 0.00 (0.000) 1
?- daa.make(real32,[2,4,2,0.5],:D), is.data(:D)
-> (  ) := 1.00 (0.000) 1
```

is.even

`is.even(`*number*`)`

The *primitive* will resolve to a *truth value* of `1` if the *term* is an even *number*, `0` otherwise. For example:

```
?- is.even(3)
-> (  ) := 0.00 (0.000) 1
?- is.even(4)
-> (  ) := 1.00 (0.000) 1
```

is.final

`is.final(`*term*`)`

The *primitive* will resolve to a *truth value* of `1` if the *term* is *final* that is isn't an unbound variable or doesn't (recursively) contains any unbound variable. For example:

```
?- is.final(5)
-> (  ) := 1.00 (0.000) 1
?- is.final([5,a])
-> (  ) := 1.00 (0.000) 1
?- is.final([5,:a])
-> ( :a ) := 0.00 (0.000) 1
```

is.func

`is.func(`*term*`)`

The *primitive* will resolve to a *truth value* of `1` if the *term* is a *functor*, `0` otherwise. For example:

```
?- is.func(66)
-> (  ) := 0.00 (0.000) 1
?- is.func(hello)
-> (  ) := 0.00 (0.000) 1
?- is.func(hello(world))
-> (  ) := 1.00 (0.000) 1
```

is.frame

`is.frame(`*term*`)`

The *primitive* will resolve to a *truth value* of `1` if the *term* is a *frame*, `0` otherwise. For example:

```
?- is.frame(hello)
-> (  ) := 0.00 (0.000) 1
?- is.frame({})
-> (  ) := 1.00 (0.000) 1
?- is.frame({a = 1, b = 2})
-> (  ) := 1.00 (0.000) 1
```

is.list

`is.list(`*term*`)`

The *primitive* will resolve to a *truth value* of `1` if the *term* is a *list*, `0` otherwise. For example:

```
?- is.list(34)
-> (  ) := 0.00 (0.000) 1
?- is.list([a,b,c,d])
-> (  ) := 1.00 (0.000) 1
```

is.number

`is.number(`*term*`)`

The *primitive* will resolve to a *truth value* of `1` if the *term* is a *number*, `0` otherwise. For example:

```
?- is.number(3)
-> (  ) := 1.00 (0.055) 1
?- is.number(hello)
-> (  ) := 0.00 (0.000) 1
```

is.odd

`is.odd(`*number*`)`

The *primitive* will resolve to a *truth value* of `1` if the *term* is an odd *number*, `0` otherwise. For example:

```
?- is.odd(3)
-> (  ) := 1.00 (0.000) 1
?- is.odd(4)
-> (  ) := 0.00 (0.000) 1
```

is.primitive

`is.primitive(`*symbol*`)`

The *primitive* will resolve to a *truth value* of `1` if the *symbol* given as *term* is the name of an existing *primitive*. `0` otherwise. For example:

```
?- is.primitive(console.puts)
-> (  ) := 1.00 (0.000) 1
?- is.primitive(print)
-> (  ) := 0.00 (0.000) 1
```

is.quirk

is.quirk(*term*)

The *primitive* will resolve to a *truth value* of `1` if the *term* is a *quirk*, `0` otherwise. For example:

```
?- is.quirk(hello)
-> (  ) := 0.00 (0.000) 1
?- is.quirk(hello^5)
-> (  ) := 1.00 (0.000) 1
```

is.range

is.range(*term*)

The *primitive* will resolve to a *truth value* of `1` if the *term* is a *range*, `0` otherwise. For example:

```
?- is.range(<1|10>)
-> (  ) := 1.00 (0.000) 1
?- is.range(231)
-> (  ) := 0.00 (0.000) 1
```

is.regexp

is.regexp(*term*)

The *primitive* will resolve to a *truth value* of `1` if the *term* is a *regexp*, `0` otherwise. For example:

```
?- is.regexp(42)
-> (  ) := 0.00 (0.001) 1
?- is.regexp('regexp("\d+"))
-> (  ) := 1.00 (0.000) 1
```

is.string

is.string(*term*)

The *primitive* will resolve to a *truth value* of `1` if the *term* is a *string*, `0` otherwise. For example:

```
?- is.string(3)
-> (  ) := 0.00 (0.000) 1
?- is.string(hello)
-> (  ) := 0.00 (0.001) 1
?- is.string("hello, world!")
-> (  ) := 1.00 (0.000) 1
```

is.symbol

`is.symbol(`*term*`)`

The *primitive* will resolve to a *truth value* of `1` if the *term* is a *symbol*, `0` otherwise. For example:

```
?- is.symbol(3)
-> (  ) := 0.00 (0.000) 1
?- is.symbol(hello)
-> (  ) := 1.00 (0.000) 1
?- is.symbol("hello, world!")
-> (  ) := 0.00 (0.000) 1
```

is.variable

`is.variable(`*term*`)`

The *primitive* will resolve to a *truth value* of `1` if the *term* is an unbound *variable*, `0` otherwise. For example:

```
?- is.variable(:h)
-> ( :h ) := 1.00 (0.000) 1
?- is.variable(5)
-> (  ) := 0.00 (0.000) 1
?- set(:h,5), !is.variable(:h)
-> ( 5 ) := 1.00 (0.000) 1
```

of.type

`of.type(`*term*`,`*variable*`|`*symbol*`)`

The *primitive* will resolve to a *truth value* of `1` if the *term* is of a given type, indicated by a *symbol* as the second *term*, `0` otherwise. For example:

```
?- of.type(4,number)
-> (  ) := 1.00 (0.000) 1
?- of.type(hello,number)
-> (  ) := 0.00 (0.000) 1
```

If the second *term* is an *unbound variable*, the type of the term will be bound to it. Possible types are: `number`, `symbol`, `string`, `binary`, `regexp`, `list`, `func`, `variable`, `frame`, `range`, `escaper`, `guid`, `quirk`, `data`.

5.19 Vector, Matrix and Quaternion

This section describes *primitives* that can be used to manipulate *lists* that represent vectors, matrices and quaternions.

mat.apply

`mat.apply(`*list*`,`*symbol*`,`*list*`,`*variable*`|`*term*`)`

The *primitive* `mat.apply` will apply the transformation described in a matrix (first *term*) to a vector (third *term*), and unifies or substitues the resulting vector to the fourth *term*. The second *term* is expected to be a *symbol* which indicate how the transformation must be applied based on the type of vector the third *term* is: `point`, `direction` or `vector`. For example:

```
?- qat.euler(:Q,[0,0,90]), mat.make([0,0,0],:Q,:M), mat.apply(:M,direction,[1,0,0],:v)
-> ( [0.000000, 1, 0] ) := 1.00 (0.001) 1
```

If the third *term* is an unbound *variable* and the fourth *term* is a *list*, the *primitive* will execute the inverse transformation:

```
?- qat.euler(:Q,[0,0,90]), mat.make([0,0,0],:Q,:M), mat.apply(:M,direction,:v,[0,1,0])
-> ( [1, 0.000000, 0] ) := 1.00 (0.000) 1
```

mat.make

`mat.make(`*list*, *list*, *variable*|*term*`)`

This *primitive* will create a matrix base transformation, using the first *term* as a translation vector and the second *term* as the rotation (quaternion). It will unifies or substitues the resulting matrix to the third *term*. For example:

```
?- qat.euler(:Q,[0,0,90]), mat.make([0,0,0],:Q,:M), mat.apply(:M,direction,[1,0,0],:v)
-> ( [0.000000, 1, 0] ) := 1.00 (0.001) 1
```

qat.add

`qat.add(`*list*, *list*, *variable*|*term*`)`

The *primitive* will add two rotations expressed in quaternions (the two first *term*) and substitues the resulting quaternion to the third *term*. For example:

```
?- qat.euler(:Q1,[0,0,90]), qat.euler(:Q2,[90,0,00]), qat.add(:Q1,:Q2,:Q), qat.euler(:Q,:e)
-> ( [0, 89.999993, 89.999993] ) := 1.00 (0.001) 1
```

qat.apply

`qat.apply(`*list*, *list*, *variable*|*term*`)`
`qat.apply(`*list*, *variable*|*term*, *list*`)`

The *primitive* will apply the rotation expressed in the quaternion (the first *term*) to a vector (the second *term*) and unifies or substitues the result vector to the third *term*. If the second *term* is an unbound *variable* and the third is a vector, the *primitive* will apply the inverse rotation. For example:

```
?- qat.euler(:Q,[0,0,90]), qat.apply(:Q,[1,0,0],:v)
-> ( [0, 1.000000, 0] ) := 1.00 (0.001) 1
?- qat.euler(:Q,[0,0,90]), qat.apply(:Q,:v,[0,1,0])
-> ( [1.000000, 0, 0] ) := 1.00 (0.001) 1
```

qat.euler

`qat.euler(`*list*, *variable*|*term*`)`
`qat.euler(`*variable*|*term*, *list*`)`

The *primitive* will extract the *Euler* angles (as a *list*, in degrees) from a quaternion expressed in a *list* and unifies or substitues the second *term* with it. If the first *term* is an unbound *variable*, the *primitive* will unify or substitue it with a quaternion build from the given *Euler* angles. For example:

```
?- qat.euler(:q,[45,0,180])
-> ( [-0.000000, -0.382683, 0.923880, -0.000000] ) := 1.00 (0.001) 1
?- qat.euler([-0.000000, -0.382683, 0.923880, -0.000000],:e)
-> ( [44.999962, 0, 180.000005] ) := 1.00 (0.000) 1
```

qat.length

`qat.length(`*list*, *variable*|*number*`)`

The *primitive* will compute the length of a quaternion (the firs *term*) and substitues the result to the second *term*. For example:

```
?- qat.euler(:Q,[180,0,90]), qat.length(:Q,:l)
-> ( 1.000000 ) := 1.00 (0.000) 1
```

qat.sub

`qat.sub(`*list*, *list*, *variable*|*term*`)`

The *primitive* will substract two rotations expressed in quaternions (the two first *term*) and substitues the resulting quaternion to the third *term*. For example:

```
?- qat.euler(:Q1,[0,0,90]), qat.euler(:Q2,[90,0,00]), qat.sub(:Q1,:Q2,:Q), qat.euler(:Q,:e)
-> ( [0, 270.000008, 90.000003] ) := 1.00 (0.001) 1
```

vec.add

`vec.add(`*list*, *list*|*number*, *variable*|*term*`)`

This *primitive* will add two vectors (the two first *term*) and unifies or substitues the result to the third *term*. If the second *term* is a *number* that value will be added to each component of the vector. For example:

```
?- vec.add([0,0,0],[1,-1,1],:v)
-> ( [1, -1, 1] ) := 1.00 (0.000) 1
?- vec.add([0,0,0],1,:v)
-> ( [1, 1, 1] ) := 1.00 (0.000) 1
```

If the second *term* is an unbound *variable* and the third *term* is a *list*, the *primitive* will unify the second *term* with the vector substraction of its third *term* with the first:

```
?- vec.add([0,0,1],:v,[3,2,1])
-> ( [3, 2, 0] ) := 1.00 (0.000) 1
```

vec.angle

`vec.angle(`*list*, *list*, *variable*|*number*`)`

This *primitive* will will compute the angle (in degree) between two given normalized direction (the first and second *terms*) and unifies or substitues it to the third *term*. For example:

```
?- vec.angle([1,0,0],[0,1,0],:v)
-> ( 90 ) := 1.00 (0.000) 1
```

vec.angle.signed

`vec.angle.signed(`*list*, *list*, *list*, *variable*`|`*number*`)`

This *primitive* will will compute the signed angle (in degree) between two given normalized direction (the first and second *terms*) given an axis (the third *term*) and unifies or substitues it to the fourth *term*. For example:

```
?- vec.angle.signed([1,0,0],[0,1,0],[0,0,1],:v)
-> ( 90 ) := 1.00 (0.000) 1
```

vec.dist

`vec.dist(`*list*, *list*, *variable*`|`*number*`)`

This *primitive* will will compute the distance between two given points (the first and second *terms*) and unifies or substitues it to the third *term*. For example:

```
?- vec.dist([0,0,0],[1,2,4],:d)
-> ( 4.582576 ) := 1.00 (0.000) 1
```

vec.div

`vec.div(`*list*, *list*`|`*number*, *variable*`|`*term*`)`

This *primitive* will dive two vectors (the two first *term*) and unifies or substitues the result to the third *term*. If the second *term* is a *number* that value will be divided from each component of the vector. For example:

```
?- vec.div([2,4,6],[2,2,2],:v)
-> ( [1, 2, 3] ) := 1.00 (0.000) 1
?- vec.div([2,4,6],2,:v)
-> ( [1, 2, 3] ) := 1.00 (0.000) 1
```

If the second *term* is an unbound *variable* and the third *term* is a *list*, the *primitive* will unify the second *term* with the inverse operation:

```
?- vec.div([2,4,6],:v,[1,2,3])
-> ( [2, 2, 2] ) := 1.00 (0.000) 1
```

vec.length

`vec.length(`*list*, *variable*`|`*number*`)`

This *primitive* will compute the length of the vector (the first *term*) and unifies or substitues with the second *term*. For example:

```
?- vec.length([2,4,6],:l)
-> ( 7.483315 ) := 1.00 (0.000) 1
```

vec.mul

`vec.mul(`*list*, *list*|*number*, *variable*|*term*`)`

This *primitive* will multiply two vectors (the two first *term*) and unifies or substitues the result to the third *term*. If the second *term* is a *number* that value will be multiplied to each component of the vector. For example:

```
    ?- vec.mul([1,2,3],[1,2,3],:v)
    -> ( [1, 4, 9] ) := 1.00 (0.000) 1
    ?- vec.mul([1,2,3],2,:v)
    -> ( [2, 4, 6] ) := 1.00 (0.000) 1
```

If the second *term* is an unbound *variable* and the third *term* is a *list*, the *primitive* will unify the second *term* with the inverse operation:

```
?- vec.mul([1,2,3],:v,[2,4,6])
-> ( [2, 2, 2] ) := 1.00 (0.000) 1
```

vec.norm

`vec.norm(`*list*, *variable*|*list*`)`

This *primitive* will normalized a vector (the first *term*) and unifies or substitues with the second *term*. For example:

```
?- vec.norm([2,4,6],:l)
-> ( [0.267261, 0.534522, 0.801784] ) := 1.00 (0.000) 1
```

vec.sub

`vec.sub(`*list*, *list*|*number*, *variable*|*term*`)`

This *primitive* will substract two vectors (the two first *term*) and unifies or substitues the result to the third *term*. If the second *term* is a *number* that value will be substracted from each component of the vector. For example:

```
?- vec.sub([0,0,0],[1,-1,1],:v)
-> ( [-1, 1, -1] ) := 1.00 (0.000) 1
?- vec.sub([0,0,0],-1,:v)
-> ( [1, 1, 1] ) := 1.00 (0.000) 1
```

If the second *term* is an unbound *variable* and the third *term* is a *list*, the *primitive* will unify the second *term* with the reverse operation:

```
?- vec.sub([0,0,1],:v,[3,2,1])
-> ( [-3, -2, 0] ) := 1.00 (0.000) 1
```

6 Elementals

This section provides some details on all the *elementals* supported by the *runtime*. For each one, the list of supported *properties* and accepted values will be given as well as some explanation on their use cases.

`MRKCAggregator`

This *elemental* provides a way to relay a query to a collection of *elementals* and aggregate the successful replies into a *list*. Not unlike `fzz.collect`, however, the *elemental* will reply as soon as it has received a reply from each of the specified labels.
This *elemental* supports the following *properties*:

`labels`	the list of all the labels (as *symbols*)
`timeout`	timeout (in seconds) in case some expected *elementals* didn't reply.

`MRKCBFSolver`

This *elemental* class is the most common one used in *fizz* . It is in fact the default and can handle *statements* as well as *prototypes*. It implement a *breadth-first* solving which is optimized for concurrency, therefore it is not the most efficient *solver* with regard to time and memory usage.

This *elemental* supports the following *properties*:

`p.limit`	the maximum number of *prototype* the object will accept when they are defined.
`s.limit`	the maximum number of *statement* the object will accept when they are asserted.
`replies.are.triggers`	set to `no` to instruct the *elemental* to not considere *replies* as potential triggers.
`memoize`	set to `yes` to instruct the *elemental* to use memoization (that is to temporary cache replies to queries in order to avoid inferring the same thing multiple time).
`reply.on`	set to `success` to instruct the *elemental* to only reply to query when successful. Set to `failure` to have it only reply on failure only.
`cascade`	when set to `yes`, the *elemental* will only jump to try another *prototype* when the previous one have failed.
`cascade.tmo`	timeout value (in seconds) before attempting another *prototype* when waiting for a reply.
`spunky`	set to `yes` to instruct the *elemental* to discard queries that it has received and solved as soon as possible. When set to `no` (the default), the *elemental* will keep the queries going for longer in case of late replies.

When such *elemental* is set to memoize, cached *statements* will be periodically cleared at a a frequency set by the `mzttl` *substrate* configuration.

`MRKCDFSolver`

This *elemental* class can handle *statements* as well as *prototypes* and implement a *depth-first* which is a more efficient *solver* than the default one, altought not always the right choice.

This *elemental* supports the following *properties*:

`tmo.first`	timeout value (in seconds) when waiting for the first response to a query initiated by the *elemental*. Default is 0.1s
`tmo.after`	timeout value (in seconds) when waiting for further response to a query initiated by the *elemental*. Default is 0.08s
`tmo.ticks`	how often the *elemental* checks if any query timedout. Default is 0.05s.

Note that *trigger predicates* are. at the moment, not supported with this *solver*.

`MRKCCatcher`

This *elemental* class can be used to catch *statements* (of the same label) that are asserted, repealed or declared and perform some inference on them.

This *elemental* supports the following *properties*:

`seize`	set to `yes` to prevent from passing along the *statements* to any other *elemental*.
`sense`	set to `yes` to catch declared *statements*.

To handle the *statements*, the definition of the *elemental* must include *prototypes* as shown in this example to accept the *queries* that will be self-generated by the *elemental* for each of the *statements*:

```
car {

    class = MRKCCatcher,
    seize = yes,
    sense = yes

} {

    (asserted,:l,:v)    :-  console.puts(:l,"=",:v," asserted");
    (repealed,:l,:v)    :-  console.puts(:l,"=",:v," repealed");
    (declared,:l,:v)    :-  console.puts(:l,"=",:v," declared");

}
```

The entrypoint of the *prototype* must unify with the *predicate* that will be generated by the *elemental*. The first *term* will be the *symbol* `asserted`, `repealed` or `declared` and the second *term* is a *list* of the *statement*'s terms. The third *term* will be the *truth value* of the statement.

`MRKCMingler`

This *elemental* class provides a way to try a set of *terms* against all possible permutation of the *terms* against a specific *knowledge*.

This *elemental* supports the following *properties*:

`query`	the label (or labels, if a *list* is provided) to be used for each out-going queries.
`arity`	the arity of the queries to be generated.

For example, if we have the following *knowledge* definition which contains some countries' flag colors:

```
flag {

    (red,white,canada);
    (red,white,japan);
    (red,yellow,china);
    (blue,white,greece);
    (blue,white,argentina);
    (blue,yellow,sweden);
    (yellow,red,spain);

}
```

Because of the way the knowledge is defined, if we wanted to test if one of the color from any of the flags is, said `yellow`, we would have to query for it in the first *term* and in the second *term*. If we use a `MRKCMingler` *elemental* as an intermediate by definiting it as such:

```
flag.any {

    class = MRKCMingler,
    query = flag,
    arity = 3

}
```

We can simply query it as follow to get the list of countries with the color yellow in their flag:

```
?- #flag.any([yellow],_,[_,_,:c])
-> ( spain ) := 1.00 (0.001) 1
-> ( china ) := 1.00 (0.001) 2
-> ( sweden ) := 1.00 (0.001) 3
```

Any query to `MRKCMingler` should have an arity of 3. The first *term* is expected a *list* or *final terms* which are expected to be in the same *statements*. The second *term*, will be unified on reply with a *list* that provides the relation between the position of the *terms* in the first *term* and the content of the *statements* received by the *elemental* as replies. If a fourth *term* is provided, it will be unified with the label of the *knowledge* that get queried.

`MRKCCSVStore`

The *elemental* `MRKCCSVStore` provides a way to access *statements* stored inside a `CSV` file without having to import the file. While this a slowest way to retreive *statements*, it has the advantage of having lower memory consumption as none of the data stored in the `CSV` file are loaded in memory until it is returned as answers to a query. The *elemental* can also accepts asserted *statements* with the right `mode`. Each new *statement* will be appended at the end of the CSV file. The default behavior is `read.only`.

This *elemental* supports the following *properties*:

`filepath`	the path and file name of the `CSV` file to be used as source.
`delimiter`	a *string* representing the character used as the column separator.
`columns`	a list describing the conversion to be applied to each of the columns that will be read from the file. The number of *terms* in the *list* is considered to be the expected number of columns in each lines of the file. If this property is not specified, each columns will be converted to best fit its content.
`offset`	the number of lines from the file to be skipped from the start of the file (e.g. to skip a header). If this property is not specified no offset will be applied.
`length`	the number of lines (from the `offset`) to be considered when scanning the file. If this property is not specified, the file will be scanned to its end.
`no.match`	if set to the *symbol* `fail`, the *elemental* will always produce a statement with a truth value of `0` when there was no match to a query.
`offloaded`	if set to the value `yes`, the scanning of the file will be offloaded to a background thread. This will lower the load on the *substrate* at the cost of a bit more lag in getting answers.
`arity`	arity of the statements (if the number of columns is greater than the arity, the extra will be grouped into a list as the last term).
`mode`	set to `read.only` to not accept any assert *statements*, set to `truncate` to truncate the file if it exists, set to `append` to accept any asserted *statements*.

The *terms* in the `columns` *list* can be any of the following:

`number`	the column is a *number*.
`symbol`	the column is a *symbol*.
`string`	the column is a *string*.
`ignore`	the colum is to be ignored.
`select`	the colum format should be selected based on the content of each line.

For example, the following *elemental* provides *statements* based on the `cars` database stored in a CSV file:

```
car {

    class       = MRKCCSVStore,
    filepath    = "./etc/data/cars.csv",
    delimiter   = ";",
    offset      = 2,
    no.match    = fail,
    offloaded   = yes

} {}
```

`MRKCProxy`

This *elemental* provides a way to relay a query to a collection of *elementals* and return only the successful replies.
This *elemental* supports the following *properties*:

`labels`	the list of all the labels (as *symbols*)
`timeout`	timeout (in seconds) in case some expected *elementals* didn't reply.
`snappy`	when set to yes (the default), the elemental will replies with failure as soon as it has received failure from all the labels.

`MRKCSBFStore`

This *elemental* provides a way to store and retreive *statements* from a *binary* file. While it is a slower way to retreive *statements*, it has the advantage of having lower memory consumption as none of the data stored in the file is loaded in memory until it is returned as answers to a query.

This *elemental* supports the following *properties*:

`filepath`	the path and file name of the `binary` file to be used as source.
`index`	the property is interpreted as the (or multiple when a *list* is given) index of the *statement*'s terms that we which the *statements* to be indexed upon. Judicious indexing will speed-up retreival of *statements*.
`no.match`	if set to the *symbol* `fail`, the *elemental* will always produce a statement with a truth value of `0` when there was no match to a query.
`offloaded`	if set to the value `yes`, access to the file will be offloaded to a background thread. This will lower the load on the *substrate* at the cost of a bit more lag in processing.
`verbose`	an optional *boolean* value (or a *symbol* `true, false`) to instructs the *elemental* to output more traces in the console.

For example, the following *elemental* setup a *statement* store in which we will import the data from `./etc/data/cars.csv`:

```
car {

    class       = MRKCSBFStore,
    filepath    = "./cars.sbfz",
    offloaded   = yes,
    index       = [0,9],
    verbose     = yes,
    no.match    = fail

} {}
```

The `/tells` console command can be used to instruct the *elemental* to perform any of the following actions:

`compact`	requests the store to attempt to reduce its file size.
`optimize`	requests the store to be optimized for better performance.
`stats`	prints some statistics about the content of the store.
`validate`	forces a sanity check on the store.
`clear`	empties the store of all stored *statements*.

Note that depending on the number of stored *statements*, many of the above command may take a while to complete.

`MRKCStopper`

This *elemental* class can be used to reply by failure to any *query* that is left un-answered for a given time. The *elemental* watches queries and wait for corresponding replies.

This *elemental* supports the following *properties*:

`qtmo`	timeout value (in seconds) when waiting for a response to a query. Default is 0.15s
`tick`	how often the *elemental* checks if any query timed-out. Default is 0.05s.
`labels`	a *list* of the *query* labels the *elemental* must watch for. If the property isn't set, the label of the *elemental* will be used.

`FZZCFUNRunner`

This *elemental* class supports mixing *imperative* style with *logic* style by providing a way to execute expressions build out of *functors* using the same *terms* commonly used in *logical statements* (which will be refered to as *f-expressions*). Code to be executed can be submitted to the *elemental* via a simple set of *query*. Each request, is considered a *thread* and will then run cooperatively with other submitted *f-expression*.

An *f-expression* can be either a *list* of *functors* to be executed sequentially or a single *functor*. Each *functor* can have for arguments other *functors* or *lists* of *functors*. The use of *variable terms* will be understood in a *f-expression* as a shortcut to using the *primitive* `get`. Finally, each *functor* is expected to return a *fizz term*. For example, the following code would print to the console the value `5`: `print(add(3,2))`.

Each *thread* is running within its own *execution context*, which means that any given *variable* is only accessible in the *thread* it was created in. Unlike in most imperative language, all *variables* in a *thread* are global even when they are first referenced within a *block*.

This *elemental* supports the following *properties*:

`bsize`	maximum number of builtin calls before a thread get interrupted. Default is 32
`functions`	label (as a *symbol*) of the query to publish when mapping a *functor* to a function.
`primitives`	label (as a *symbol*) of the query to publish when mapping a *functor* to a primitive.

As an alternative to using an *elemental* to define *functions*, the *elemental* itself can contains *functions* definitions. In that case, specifying a value for the `functions` *property* is not necessary.

The *elemental* will reply to the following queries:

`eval`	execute some code (the 2nd *term*) synchronously and unify the result with the 3rd *term*	`#funx(eval,add(4,:x),:y)`
`start`	execute some code (the 2nd *term*) asynchronously and unify the 3rd *term* with a *term* that uniquely identify the *thread*	`#funx(start,do(something),:uuid)`
`stop`	stop the execution of a *thread* identified by its identifier given as the 2nd *term*.	`#funx(stop,:uuid)`
`cancel`	cancel the execution of a *thread* identified by its identifier given as the 2nd *term*.	`#funx(cancel,:uuid)`
`list`	unify the 2nd *term* with a *list* of all *threads* currently running	`#funx(list,:l)`
`value`	unify the 3rd *term* with the value returned by *thread* identified by its *identifier* (the 2nd *term*). When a *thread* is still running, its *value* will be `nil`.	`#funx(state,:uuid,:v)`
`state`	unify the 3rd *term* with the state of a *thread* identified by its *identifier* (the 2nd *term*). The state will either be `executing` or `completed`	`#funx(state,:uuid,:s)`
`send`	send the 3rd *term* to the *thread* identified by its *identifier* (the 2nd *term*).	`#funx(send,:uuid,[1,2,3])`

`state` and `value` queries won't be answered if the *thread* is unknown. The *time-to-live* value of the *elemental* will be used to determinate when a completed *thread*'s value can be forgotten.

f-expressions' functor can be calls to known *builtins*, *primitives* or *functions*. The difference is minimum, but important. *Builtins* are executed directly by the *elemental* without need for a *query* to be sent out. Here's a list of all the supported *builtins*:

`is.atom(term)`	return `1` if *term* is an *atom*, `0` otherwise.
`is.number(term)`	return `1` if *term* is a *number*, `0` otherwise.
`is.string(term)`	return `1` if *term* is a *string*, `0` otherwise.
`is.symbol(term)`	return `1` if *term* is a *symbol*, `0` otherwise.
`is.binary(term)`	return `1` if *term* is a *binary*, `0` otherwise.
`is.list(term)`	return `1` if *term* is a *list*, `0` otherwise.
`is.func(term)`	return `1` if *term* is a *functor*, `0` otherwise.
`is.frame(term)`	return `1` if *term* is a *frame*, `0` otherwise.
`is.range(term)`	return `1` if *term* is a *range*, `0` otherwise.
`is.regexp(term)`	return `1` if *term* is a *regexp*, `0` otherwise.
`is.guid(term)`	return `1` if *term* is a *guid*, `0` otherwise.
`is.quirk(term)`	return `1` if *term* is a *quirk*, `0` otherwise.
`is.data(term)`	return `1` if *term* is a *data*, `0` otherwise.
`var.capture(labels,mode)`	capture all or some of the *variables* in the *thread* into a *frame*. If *mode* is `inclusive`, *labels* will be assumed to be a list of all the *variables* to capture. If it is `exclusive`, the call will capture all *variables* but the one listed in *labels*.
`var.release(frame)`	take the *frame* passed as argument and use each key/value pairs as a *variable* to be set.
`is.canceled()`	test if the thread has been canceled.
`recv()`	read the next *term* that was sent (with the `send` command) to the thread. Returns `nil`if there is no *term* available.
`peek(label,term?)`	access the *thread local data* and retreive a previously stored value by its *label*. If a *term* is provided, it will be returned if the identifier is unknown.
`poke(label,value)`	access the *thread local data* and store a *value* (2nd argument) for a given *label*.
`zero(label)`	remove a value from the *thread local data* given its *label*.
`cls(label+)`	unset given *variable(s)*.
`set(label,value)`	set the *value* of a *variable* identified by its *label*. The call will return the *value*.
`get(label)`	return the value of a given *variable*.
`inc(label)`	increase the value stored in the *variable* identified by its *label* by `1`.
`dec(label)`	decrease the value stored in the *variable* identified by its *label* by `1`.
`eq(term,term)`	return `1` if its two *terms* are equal, `0` otherwise.
`neq(term,term)`	return `0` if its two *terms* are equal, `1` otherwise.
`gt(term,term)`	return `1` if the first *term* is greater than the 2nd, `0` otherwise.
`gte(term,term)`	return `1` if the first *term* is greater-or-equal to the 2nd, `0` otherwise.
`lt(term,term)`	return `1` if the first *term* is lesser than the 2nd, `0` otherwise.
`lte(term,term)`	return `1` if the first *term* is lesser-or-equal to the 2nd, `0` otherwise.
`and(term+)`	return the *boolean and* of all its arguments.
`or(term+)`	return the *boolean or* of all its arguments.
`xor(term+)`	return the *boolean xor* of all its arguments.
`not(term)`	return the *boolean not* of *term*.
`add(term+)`	return the sum of all its arguments.
`sub(term+)`	return the subtraction of all its arguments.
`mul(term+)`	return the multiplication of all its arguments.
`div(term+)`	return the division of all its arguments.
`div.int(term+)`	return the integer division of all its arguments.
`lst.append(list,term)`	return a new *list* with a *term* appended to the first term.
`lst.empty(list)`	return `1` if the *term* is an empty *list*.
`lst.prepend(list,term)`	return a new *list* with a *term* prepended to the first term.
`lst.item(list,index)`	return the item at a given *index* in a *list*.
`lst.length(list)`	return the number of items in a *list*.
`lst.head(list)`	return the first item in the *list*, or `nil` if empty.
`lst.tail(list)`	return the last item in the *list*, or `nil` if empty.
`lst.rest(list)`	return a new *list* minus the first item in the *list*.
`lst.make(term+)`	return a *list* of all the arguments.
`frm.labels(frame)`	return a list of all the labels in the frame.
`frm.fetch(frame,label,term?)`	return the value associated with a given *label* in a *frame*. If the *label* doesn't exists, `nil` will be returned unless *term* is provided.
`frm.store(frame,label,value)`	return a new *frame* with a new *value* associated with a given *label*.

`sleep(time)`	put the calling *thread* to sleep for the specified *time* (in ms).
`publish(label,terms,value)`	publish a *statement* built from a *label*, a list of *terms* and a truth *value*.
`await(label,timeout)`	block and wait for any *statement* to be published with the given *label* or until a *timeout* (in ms). The result will be the *list* of terms, or the value `0` if timeout occured.
`call(label,terms)`	takes the *label* of a *functor* to be executed a list of *terms* to be passed to it and execute it, then returns whatever that call returned.
`eval(expr)`	evaluate the expression then execute whatever it returned as instructions.
`yield()`	force the calling *thread* to yield to any other concurrently executing *threads*.
`times(functor+)`	time the execution of its argument and return the elapsed time (in s).
`self.peek(label,term?)`	access the *elemental's properties* and retreive a previously stored value by its *label*. If a *term* is provided, it will be returned if the identifier is unknown.
`self.poke(label,value)`	access the *elemental's properties* and store a *value* (2nd argument) for a given *label*.
`self.zero(label)`	remove a value from the *elemental's properties* given its *label*.
`inquire(label,terms,v?,c?,n?)`	post a logical query built from a *label* and a list of *terms* and execute *c* for each of the replies the query will generate. If a 5th term is provided, it is assumed to be a timeout value in seconds. The call will return the number of *statements* that were received and for each reply, the variable *v* will be set to the truth value of the statement. If only two of the *terms* are given, the call will not wait for any reply. If *label* is `self`, the query will be addressed to the *elemental* itself.
`unify(term,term)`	perform a *logical unification* between the two *terms* and return `1` if it succeeded and `0` if not.

When it comes to *control structure*, *f-expressions* supports the following:

`quote(expr)`	protect *expr* from evaluation.
`if(expr,left,right)`	if *expr* evaluate to `1`, the *left* instruction will be executed, otherwise *right* one will be.
`do(code,expr)`	will execute the *code* instruction and execute again as long as *expr* evaluate to `1`.
`while(expr,code)`	will execute the *code* instruction as long as *expr* evaluate to `1`.
`loop(count,code)`	will execute the *code* instruction as many times as requested with the value *count*.
`break()`	will break the execution of any loop/do/while/switch.
`return(value)`	will cause the execution of any thread/function to stop and return the given *value*.
`result(value)`	if the *thread* was started to be synchronous, the call will provides an answer to it without causing the *thread* to end.
`foreach(var,list,code)`	will execute the *code* for each item in the *list*, and set the *variable* for which the label was specified (*var*) to the item at each loop.
`switch(value,block)`	will execute a list of instruction where each `case()` will compare to the *value*.
`case(value,code)`	will execute the *code* if *value* match the parent `switch()`'s *value*.
`retry(code,value)`	will keep executing the *code* as long it return *value*.

The way you can extend the capabilities of *f-expression* is by defining *primitives* and *functions*. *Primitives* are a way to connect an *imperative* execution to a *logical* query, with the caveat that the *f-expression* will only accept the first answer to a *primitive* leading to a query. To define a *primitive*, you simply as to create an *elemental* with the same label as the one specified in the *elemental*'s property `primitives` and add one (or more) *prototype*(s) per primitive. Note that the *entrypoint* of the *prototype* must have an arity of three and be: the label of the primitive, a *list* accepting the arguments to the call, the return value. Here's an example where we define the `car` and `cdr` *primitives*:

```
funx.primitive {

    (car,[[:h|_]],:h)^           :-  true;
    (cdr,[[_|:r]],:r)^           :-  true;

}
```

Altough the example above is simplistic, note that a *primitive* can be implemented by any *logical* combination that may be necessary.

Functions can be defined in a similar way than *primitives* are, except they are not implemented as *prototypes* but as *statements*. And thus, they are queried only once and the body of the *function* is cached. Like the *primitives*, each *functions* must be defined in an *elemental* with for label the one specified in the `functions` property. Each *statement* must have an arity of three: the label of the function, a *list* of *symbols* that are the *variables* to get assigned the arguments passed to the call, and a *list* of *instructions*. Here's an example of a *function* that will compute the sum of a *list* using the `foreach()` *instruction*:

```
funx.function {

    (sum,[l],[
        set(s,0),
        foreach(v,:l,[
            set(s,add(:s,:v))
        ]),
        return(:s)
    ]);

}
```

A cached *function* will be invalidated, if they may have been replaced (e.g. by reloading the file in which the *elemental* in which it resides). Also note, that *primitives* and *functions* can be defined by any number of *elementals*. For examples of *f-expressions*, see the samples `funx.fizz`, `funx2.fizz` and `funx3.fizz`.

There is three new *constants* that can be queried during the execution of the *f-expression*: `$path`, `$uid` and `label`. The former will return an *atom* that uniquely identify the *function* (taking the call stack into action) in which the call is made. The second will returns the unique identifier of the *thread* executing the code, and the latter will provide the name of the executing *function* (or `nil`) if the calling code is running outside of a *function*.

`FZZCRandomizer`

This *elemental* can be used to inject some random activations by firing *statements* with a random *number* or *term* at a given interval. For example, we can define such *elemental* and instruct it to pick a random number between `1550` and `1650`:

```
rand {
    class = FZZCRandomizer,
    min   = 1550,
    max   = 1670,
    mod   = 2
} {

}
```

If we then load it in the *runtime* environment, it will starts firing at regular interval (the `mod` value indicates every other interval). If we use the `/spy` command, we can observe the generated *statements* being broadcasted through the *substrate*:

```
?- /spy(append,rand)
spy : observing rand
```

```
spy : S rand(1637) := 1.00
spy : S rand(1643) := 1.00
spy : S rand(1576) := 1.00
spy : S rand(1610) := 1.00
spy : S rand(1608) := 1.00
spy : S rand(1597) := 1.00
spy : S rand(1636) := 1.00
spy : S rand(1618) := 1.00
spy : S rand(1563) := 1.00
spy : S rand(1565) := 1.00
```

If we now make use of a `rand` *predicate* in a *prototype* as follows:

```
male {

    (james1, 1566)    := 1.0;
    (charles1, 1600)  := 1.0;
    (charles2, 1630)  := 1.0;
    (james2, 1633)    := 1.0;
    (george1, 1660)   := 1.0;
    (_,_)             := 0.0;

}

dad {

    (:x) :- @rand(:y) , #male(:x,:y);

}
```

We will activate a query on the `male` *predicate* each time a new `rand` *statement* is broadcasted as we can see below:

```
?- /spy(append,rand,dad)
spy : observing rand
spy : observing dad
spy : S rand(1627) := 1.00
spy : S rand(1580) := 1.00
spy : S rand(1618) := 1.00
spy : S rand(1571) := 1.00
spy : S rand(1654) := 1.00
spy : S rand(1630) := 1.00
spy : S dad(charles2) := 1.00
spy : S rand(1622) := 1.00
spy : S rand(1579) := 1.00
spy : S rand(1582) := 1.00
spy : S rand(1632) := 1.00
spy : S rand(1617) := 1.00
spy : S rand(1566) := 1.00
spy : S dad(james1) := 1.00
spy : S rand(1598) := 1.00
spy : S rand(1663) := 1.00
spy : S rand(1666) := 1.00
```

If the `min` and `max` *properties* are not specified, the *elemental* will generate random *numbers* between `0` and `1`. If only the minimum value is omitted, it will default to `0`. If it is the maximum value that is missing, it will default to the maximum possible value for a floating point *number*.

Instead of generating *number*, we can instructs the *elemental* to randomly pick an element from a *list*. To do that, we simply specify the *list* using the label `values` in the *properties*. Here's the *elemental* we used earlier rewritten to restrict the possible *numbers*:

```
rand {
    class = FZZCRandomizer,
    values = [1566,1600,1630,1633,1660]
} {

}
```

This time around, since we are only picking from the years present in the `male` *knowledge* we get `dad` *statements* right away:

```
?- /spy(append,rand,dad)
spy : observing rand
spy : observing dad
spy : S rand(1566) := 1.00
spy : S dad(james1) := 1.00
spy : S rand(1600) := 1.00
spy : S dad(charles1) := 1.00
spy : S rand(1633) := 1.00
spy : S dad(james2) := 1.00
spy : S rand(1633) := 1.00
spy : S dad(james2) := 1.00
spy : S rand(1630) := 1.00
spy : S dad(charles2) := 1.00
spy : S rand(1566) := 1.00
spy : S dad(james1) := 1.00
spy : S rand(1600) := 1.00
spy : S dad(charles1) := 1.00
spy : S rand(1633) := 1.00
spy : S dad(james2) := 1.00
```

`FZZCTicker`

This *elemental* can be used to activate other *elemental* at a regular interval by firing a *statement*. For example:

```
tick {
    class = FZZCTicker,
    mod   = 4
} {

}
```

If we then load it in the *runtime* environment, it will starts firing at regular interval (the `mod` value indicates how often based on the *substrate*'s pulse). If we use the `/spy` command, we can observe the generated *statements* being broadcasted through the *substrate*:

```
?- /spy(append,tick)
spy : observing tick
spy : S tick(9, 1512157341.254642) := 1.00 (15.000000)
spy : S tick(10, 1512157342.254716) := 1.00 (15.000000)
spy : S tick(11, 1512157343.254030) := 1.00 (15.000000)
spy : S tick(12, 1512157344.254033) := 1.00 (15.000000)
spy : S tick(13, 1512157345.253880) := 1.00 (15.000000)
spy : S tick(14, 1512157346.254291) := 1.00 (15.000000)
spy : S tick(15, 1512157347.254672) := 1.00 (15.000000)
```

The first *term* in the published *statement* is a cycle counter (which will be saved by the *elemental* when it is saved or frozen). The second *term* is the current time (in seconds since Epoc, GMT). Instead of basing the ticking on the *substrate*'s pulse, the property *tick* can be used to indicate the interval in seconds. For example, to have the *tick* statement firing every 1.5 seconds, we would write:

```
tick {
    class = FZZCTicker,
    tick  = 1.5
} {

}
```

`MRKCLettered`

The `MRKCLettered` *elemental* can only handle *statements*. It is meant to be used as a way to lower *runtime* cost when it is known that a particular *Knowledge* will never contains any *prototypes*. Here are the *properties* specific to this class:

`s.limit`	the maximum number of *statement* the object will accept when they are asserted.
`no.match`	if set to the *symbol* `fail`, the object will always produce a statement with a truth value of `0` when there was no match to a query.
`index`	the property is interpreted as the (or multiple when a *list* is given) index of the *statement*'s terms that we which the *statements* to be indexed upon. Judicious indexing will speed-up retreival of *statements* (see the sample `cars.fizz` for an example).
`loose`	if set to the *symbol* `yes`, the object will not favor the final *statements* over the non-final ones.
`nearest.only`	if set to the *symbol* `yes`, the object will always answers queries with *constrained variables* using the primitive `aeq` with the closest match possible.
`recall.frq`	how often to check stored statements for possible ones to purge.
`recall.ttl`	initial time-to-live value for any asserted statements.
`recall.add`	how much to add to a statement time-to-live each time it is used in a reply.
`recall.mul`	how much to increase (ttl + mul * ttl) to a statement time-to-live each time it is used in a reply.
`recall.thd`	threshold for committing statement to permanent storaga.

In order for the *recall* ability of the class to work. The *statements* must include a property called `stp` which contains the timestamp of the *statement* (assigned to `%now` for example when the *statement* is created). As long as the timestamp of the *statement* plus its `time-to-live` is after the current time each time the *elemental* checks, the *statement* will be conserved. Otherwise, it will be removed.

7 Modules

This section provides some details on all the optional *modules* that can be loaded in the *runtime* and which *elementals* they provides. Like in the previous section, the list of supported *properties* and accepted values will be given as well as some explanation on their use cases. In order to be able to use the *elementals* detailed in this section, the corresponding module in which it resides must be loaded in *fizz* using either the `/use` *command* or via a *solution* file.

LGR

The `modLGR` module provides an interface to the *Link Grammar Parser*[3] by the Carnegie Mellon University. It is a syntactic parser for (mainly) English sentences. The integration of the parser to *fizz* allows for a *string* to be parsed and its syntactic components bto e made available in a series of *lists*.

`FZZCLGRProcessor`

This *elemental* is the main interface to *Link Grammar Parser*. It supports the following *properties*:

`datapath`	the path to the root folder containing the parser's data. A version of it is included in fizz in `etc/data/lgr`
`language`	the language to be parsed by the *elemental*. At this moment, only English (`us` is supported.
`load.on.attach`	when set to `yes`, the *elemental* will preload the parser data when it is attached to the *substrate*. Otherwise, it will wait the first *query* to do so.

Let's look at an example (for more details, check the sample `etc/samples/linkg.fizz`). In a new *fizz* source file, we add the following:

```
lgr.parse {
    class           = FZZCLGRProcessor,
    datapath        = "./etc/data/lgr",
    language        = "us",
    load.on.attach  = yes
} {}
```

The expected arity for any query to the *elemental* we have now created in the *substrate* is five. The first term is a *frame* containing options, the second term is the the *string* to be parsed followed by four unbound *variables*:

```
?- #lgr.parse({},"the quick brown fox jumps over the lazy dog.",:ws,:ls,:ln,:cn)
-> ( [[[], nil], [[], "the"], [[a], "quick"], [[a], "brown"], [[n], "fox"], [[v], "jumps"], [[], "
    over"], [[], "the"], [[a], "lazy"], [[n], "dog"], [[], "."], [[], nil]] , [[X, [p], 0, 10], [WV
    , [], 0, 5], [W, [d], 0, 4], [S, [s, s], 4, 5], [D, [s, x], 1, 4], [A, [], 2, 4], [A, [], 3,
    4], [MV, [p], 5, 6], [J, [s], 6, 9], [D, [s, x], 7, 9], [A, [], 8, 9], [RW, [], 10, 11]] , [0,
    [1, [2, 0, [3, [4, 1, [5, 2, [6, 3, 4]]], [7, 5, [8, 6, [9, 7, [10, 8, 9]]]]]]], 5], [11, 10,
    11]] , [S, [[NP, [1, 2, 3, 4]], [VP, [5, [PP, [6, [NP, [7, 8, 9]]]]]], 10]] ) := 1.00 (0.011) 1
```

The first *variable* will be unified with the list of all the *words* which have been detected in the sentence. The second *variable* will be unified with the list of all *links* (that is the relationships between *words*). The third *variable* will unify with a *tree* describing how the sentence is structured. The fourth, and final, *variable* will

[3] http://www.link.cs.cmu.edu/link/

be unified to a *tree* which describes the components in the sentence as generated by the *Phrase Parser*[4].

The options one can provides optionally when parsing a sentence are the following:

`count`	the maximum number of alternative parsing solution to be returned (default is `1`).
`cost`	how the cost of each linkage influence what is returned: `low`: sort the linkage to have the less costly first `high`: sort the linkage to have the more costly first `first`: the linkages are in the order provided by *link grammar*

We will now defines the contents of each of the *list*, starting with the *words list*:

```
?- #lgr.parse({},"the quick brown fox jumps over the lazy dog.",:ws,_,_,_)
-> ( [[[], nil], [[], "the"], [[a], "quick"], [[a], "brown"], [[n], "fox"], [[v], "jumps"], [[], "
    over"], [[], "the"], [[a], "lazy"], [[n], "dog"], [[], "."], [[], nil]] ) := 1.00 (0.021) 1
```

Each of the *word* is described as a *list* containing first a *list* of symbols which the parser calls *subscripts*, followed by the actual *word*. In most cases, the *word* is represented as a *string*, except when the *word* isn't really a word, but what the parser calls `LEFT-WALL` or `RIGHT-WALL` (that is the start or the end of the sentence). In this example, the *word* `brown` is flagged with `a` indicating that it is an *adjective* where the *word* `jumps` is flagged with a `v` as it is a *verb*.[5]

```
?- #lgr.parse({},"the quick brown fox jumps over the lazy dog.",_,:ls,_,_)
-> ( [[X, [p], 0, 10], [WV, [], 0, 5], [W, [d], 0, 4], [S, [s, s], 4, 5], [D, [s, x], 1, 4], [A,
    [], 2, 4], [A, [], 3, 4], [MV, [p], 5, 6], [J, [s], 6, 9], [D, [s, x], 7, 9], [A, [], 8, 9], [
    RW, [], 10, 11]] ) := 1.00 (0.011) 1
```

The second *list* contains all the *links* that compose the parsed sentence. Each of which is described by a *list* containing four *terms*. The first one is a *symbol* representing the *link-type*[6], followed by a *list* of the *subscripts*. The third and fourth *terms* in the *list* are the index (in the *words list*) of the *words* that are associated with the *link*.

```
?- #lgr.parse({},"the quick brown fox jumps over the lazy dog.",_,_,:ln,_)
-> ( [0, [1, [2, 0, [3, [4, 1, [5, 2, [6, 3, 4]]], [7, 5, [8, 6, [9, 7, [10, 8, 9]]]]]], 5], [11,
    10, 11]] ) := 1.00 (0.011) 1
```

The third *list* contains how the *links* are connected into a *tree* describing the structure of the sentence. Each of the *sub-lists* is composed of three *terms*, the first one being the index of the *link* in the links *list*. The second and third *terms* can either be the index of the *word* or another node in the *tree*.

```
?- #lgr.parse({},"the quick brown fox jumps over the lazy dog.",_,_,_,:cn)
-> ( [S, [[NP, [1, 2, 3, 4]], [VP, [5, [PP, [6, [NP, [7, 8, 9]]]]]], 10]] ) := 1.00 (0.010) 1
```

Thr fourth *list* is a *Penn tree-bank* style phrase structure (a tree). Each *lists* that forms the tree has two *terms*. The first one is a *Penn type* (as a *symbol*) and the second one is a *list*. Each *terms* in that *list* can either be the index of the *word* or a *list* describing a new *Penn type* node of the tree.

[4] http://www.link.cs.cmu.edu/link/ph-explanation.html
[5] see section 3.3 in https://www.abisource.com/projects/link-grammar/dict/introduction.html for a list of the subscripts
[6] see https://www.abisource.com/projects/link-grammar/dict/index.html for details

WWW

The `modWWW` module provides ways for *fizz* to fetch data from existing `REST` services.

`FZZCWebAPIGetter`

The `FZZCWebAPIGetter` *elemental* performs a (GET) connection to a specific HTTP web service in order to respond to a received query. Part of the query will be used to compose the URL. When the service replies, the JSON document will be parsed and its content converted into a *frame*.

The *elemental*'s properties are the following:

`headers`	an optional *frame* describing all the headers to be added to the request
`flags`	a set of *symbols* modifying the behavior of the JSON to *frame* convertor. The flag `stringify` will keep all strings as *string terms*, `symbolize` will force all strings to be converted as *symbols*. The default behavior is to convert the strings that can be considered *symbol* as such
`url.host`	the scheme and hostname of the web service (http or https)
`url.path`	the path of the requested resource
`url.escape`	an optional *boolean* value (or a *symbol* `true, false`) to instructs the *elemental* to not escape the query *string*.
`verbose`	an optional *boolean* value (or a *symbol* `true, false`) to instructs the *elemental* to output more traces in the console

For example, to *get* any conversion rate from `api.fixer.io`, we would define the *elemental* as follow:

```
fixer.get {

    class           = FZZCWebAPIGetter,
    url.host        = "http://api.fixer.io",
    url.path        = "/latest",

} {

}
```

Whenever we want to query the latest conversion for said, the US Dollar, we would query it as such:

```
?- #fixer.get({ base = USD },:l)
-> ( [1525392000, 200, {Date = "Sun, 06 May 2018 02:40:35 GMT", Connection = "keep-alive",
Set-Cookie = "__cfduid=d70c6e9991dfeb2ae1ee6e8293a1622341525574434; expires=Mon, 06-May-19
02:40:34 GMT; path=/; domain=.fixer.io; HttpOnly", Cache-Control = "public, must-revalidate,
max-age=900", Last-Modified = "Fri, 04 May 2018 00:00:00 GMT", X-Deprecation-Message = "This
API endpoint is deprecated and will stop working on June 1st, 2018. For more information please
visit: https://github.com/fixerAPI/fixer#readme", Vary = "Origin", X-Content-Type-Options =
"nosniff", Server = "cloudflare", CF-RAY = "41681539a27192f4-SJC"}, {a__deprecation_message__ =
"This API endpoint is deprecated and will stop working on June 1st, 2018. For more information
please visit: https://github.com/fixerAPI/fixer#readme", base = USD, date = "2018-05-04",
rates = {AUD = 1.329700, BGN = 1.634100, BRL = 3.546300, CAD = 1.287500, CHF = 0.998410,
CNY = 6.359200, CZK = 21.308000, DKK = 6.223700, EUR = 0.835490, GBP = 0.737200,
HKD = 7.849600, HRK = 6.186000, HUF = 262.240000, IDR = 13978, ILS = 3.621200,
INR = 66.862000, ISK = 102.100000, JPY = 108.920000, KRW = 1076.400000, MXN = 19.156000,
MYR = 3.938000, NOK = 8.057500, NZD = 1.425900, PHP = 51.673000, PLN = 3.554400, RON = 3.895100,
RUB = 63.065000, SEK = 8.832800, SGD = 1.333600, THB = 31.755000, TRY = 4.257900,
ZAR = 12.628000}}] ) := 1.00 (0.400) 1
```

The *list* unified with the *variable* `:l` will contains four *terms*: a time stamp (UTC, expressed in seconds since Unix epoch), an HTTP response status number (200 for Okay), a *frame* containing the response *headers* received from the web site and a *frame* containing the data received as response.

`FZZCWebAPIPoster`

The `FZZCWebAPIPoster` *elemental* performs a (PUT) connection to a specific HTTP web service in order to respond to a received query. Part of the query will be the content to be posted to the service. When the service replies, the JSON document will be parsed and its content converted into a *frame*. If the *Content-Type* is set to `multipart/form-data`, the f*frame* will be interpreted as specifying a mime part body data. In which case, the following key must be present in the *frame*: `name`, `type`, `filename` and `data`. The later, must have for value a *binary* term.

The *elemental*'s properties are the following:

`headers`	an optional *frame* describing all the headers to be added to the request
`flags`	a set of *symbols* modifying the behavior of the JSON to *frame* convertor. The flag `stringify` will keep all strings as *string terms*, `symbolize` will force all strings to be converted as *symbols*. The default behavior is to convert the strings that can be considered *symbol* as such
`url.host`	the scheme and hostname of the web service (http or https)
`url.path`	the path of the requested resource
`verbose`	an optional *boolean* value (or a *symbol* `true, false`) to instructs the *elemental* to output more traces in the console

For example, to *get* some string analyzed for its overall sentiment using `https://sentim-api.herokuapp.com/` we would define the *elemental* as follow:

```
sentiment.api {
    class    = FZZCWebAPIPoster,
    url.host = "https://sentim-api.herokuapp.com",
    url.path = "/api/v1/",
    headers  = {Accept = "application/json", "Content-Type" = "application/json"}
}
```

Whenever we want to analyze some text, we would use the following *elemental*:

```
sentiment {

    (:s?[is.string],:R)^    :- #sentiment.api({text = :s},[_,_,_,{sentences = :r}]), any(:r,[],:R);
    (_,[])                  :- true;

}
```

The first *term* is a frame describing the content to be sent (it will be converted to JSON) and the the second *term* will on result be unified to a *list* containing four *terms*: a time stamp (UTC, expressed in seconds since Unix epoch), an HTTP response status number (200 for Okay), a *frame* containing the response *headers* received from the web site and a *frame* containing the data received as response.

FZZCWebAPIPuller

The `FZZCWebAPIPuller` *elemental* handles a temporary (but repeatable) connection to an HTTP web service, from which data (in JSON format) are to be retreived. When the JSON document received as reply has been parsed, its content will be converted into a *frame*, and the *elemental* will publish a statement containing it. In order to be able to use this *elemental*, the module in which it resides (`modWWW`) must be loaded in *fizz* using either the `/use` *command* or a *solution* file.

The *elemental*'s properties are the following:

`tick`	the frequency (in seconds) at which the web service is to be pulled. When that property isn't set, the *elemental* will only fetch the data once
`headers`	an optional *frame* describing all the headers to be added to the request
`flags`	a set of *symbols* modifying the behavior of the JSON to *frame* convertor. The flag `stringify` will keep all strings as *string terms*, `symbolize` will force all strings to be converted as *symbols*. The default behavior is to convert the strings that can be considered *symbol* as such
`url`	a single string containing the URL of the requested service/path/query, or:
`url.host`	the scheme and hostname of the web service (http or https)
`url.path`	the path of the requested resource
`url.query`	a frame describing the query, each of the label/value pair will be concatenated into a query string
`verbose`	an optional *boolean* value (or a *symbol* `true, false`) to instructs the *elemental* to output more traces in the console

For example, to *pull* the conversion USD conversion rate from `api.fixer.io`, we would have:

```
web.conv.puller {

    class           = FZZCWebAPIPuller,
    tick            = 60.0,
    url.host        = "http://api.fixer.io",
    url.path        = "/latest",
    url.query       = { base = USD }

} {

}
```

The *statement* published at each successful *pull*, will have four *terms*: a time stamp (UTC, expressed in seconds since Unix epoch), an HTTP response status number (200 for Okay), a *frame* containing the response *headers* received from the web site and a *frame* containing the data received as response. For the example above, a possible *statement* will be:

```
web.conv.puller(1518998400, 200, {Server = "nginx/1.13.8", Date = "Tue, 20 Feb 2018 04:44:55 GMT",
Connection = "keep-alive", Cache-Control = "public, must-revalidate, max-age=900",
Last-Modified = "Mon, 19 Feb 2018 00:00:00 GMT", Vary = "Origin",
X-Content-Type-Options = "nosniff"}, {base = USD, date = "2018-02-19", rates = {AUD = 1.263200,
BGN = 1.576000, BRL = 3.233400, CAD = 1.256400, CHF = 0.927720, CNY = 6.344400, CZK = 20.409000,
DKK = 6.001600, EUR = 0.805800, GBP = 0.713860, HKD = 7.822300, HRK = 5.994000, HUF = 250.730000,
IDR = 13553, ILS = 3.519200, INR = 64.253000, ISK = 100.480000, JPY = 106.560000, KRW = 1066.900000,
MXN = 18.544000, MYR = 3.890500, NOK = 7.782000, NZD = 1.355400, PHP = 52.458000, PLN = 3.340900,
RON = 3.756100, RUB = 56.463000, SEK = 7.989900, SGD = 1.313100, THB = 31.380000, TRY = 3.753000,
ZAR = 11.653000}})
```

FZZCWebURLGetter

The FZZCWebURLGetter *elemental* performs a download orented (GET) connection using an URL as a query.

The *elemental*'s properties are the following:

headers	an optional *frame* describing all the headers to be added to the request
verbose	an optional *boolean* value (or a *symbol* true, false) to instructs the *elemental* to output more traces in the console

Queries are expected to takes 3 *terms*: a *string* containing the URL of the file to download, a *string* containing the path of the file to copy the downloaded data into. The 3rd and last *term* will be unifed with a *list* containing a timestamp, the HTTP status and the headers we got from the server.

CLU

The modCLU module supports the building of a *cluster* from instances of *fizz* running on several computers (on the same network). This use a custom protocol build on top of UDP (multicast and unicast). While transmission between multiple hosts isn't garanteed to be delivered, the protocol does account for packet losses and will attempt to resend packets when needed.

FZZCCLUGateway

FZZCCLUGateway is the *elemental* which provides a link between the local instance of *fizz* and the cluster. For a cluster to work, most of the *properties* specific to this class must be identical on each instances. They are:

MCAddress	Multicast group address (as a string)
TXUDPPort	Multicast group UDP port (for message)
CLUDPPort	Multicast group UDP port (for control)
CLCadence	Multicast group heartbeat frequency (in ms)
MCTimeout	missing peer timeout (in ms)
XXTimeout	RX/TX timeout (in ms per packet)
TXTimeout	how long to keep a transmission around (in ms) for possible resends
SyCadence	sync timestamp frequency (in ms, 0 for off)
TXCadence	transmission frequency (in ms)
Bandwidth	bandwidth restriction (in byte per ms)
TXSpacing	interval of time between two consecutive transmissions (0 for none, the value is assumed to be for a TX at capacity)
PkBLength	usable UDP packet length (in bytes)
PkRetries	missing packets retry count (0 for no limit)
PkWinSize	size of the sliding window used to determinate if packets are considering missing (0 for default)
RXCadence	maximum elapsed time to spend processing received packets in one batch (in ms, 0 for no limit)
filters	a list of the *statements* and *predicates* labels to be accepted for incoming and outgoing transmissions

Default values will be used for most of the *properties* except: MCAddress, TXUDPPort and CLUDPPort. It is highly recommended to also provide a value for Bandwidth. It indicates how much of the bandwidth can be used to send data, taking in consideration the medium (e.g. WiFi vs GigE) and the number of computers that compose the cluster. For example, for 3 instances connected via a 100Mbps Ethernet, we would take the theoretical bandwidth value of 12500 bytes per milliseconds and divide it by 3. To that we can also take away a certain percentage (said 5) of it to account for other traffic, resulting in a value of 3958 bytes. When it comes to minimizing lost packets, the receive and send buffers of the computers may need to be adjusted.

As an alternative to tweaking the *Bandwidth* value yourself, you can use the following three properties to specify the cluster's setup:

`Bandwidth.value`	bandwidth available for the cluster (in byte per ms)
`Bandwidth.peers`	number of peers in the cluster
`Bandwidth.limit`	percentage of the bandwidth to reserve for the cluster usage

MLK

The `modMLK` module provides *elementals* dealing with *Machine Learning* tasks.

FZZCFFBNetwork

The `FZZCFFBNetwork` *elemental* manages a collection of *feed-forward back propagation neural networks* all built from the same training data whose are collected by querying the *runtime environment*. Once they have been trained, the *elemental* can be used for *classification* as well as *regression*. From *runtime* session to session, the trained models can be saved as part of the *properties*.

In order to be usable, this *elemental* requieres various values to be provided in its *properties*. The following table contains them:

`query`	the *predicate* (in the form of a *functor*) to be used to query for *statements* to be used as training data.
`generalize`	a *list* of *lists* describing which of the *statements terms* will be considered an input or an output.
`formatting`	a *list* describing how each of the *terms* in a *statement* is to be understood (data or label).
`hidden_layers`	a *number* providing the number of hidden layers to be used by the *neural networks*.
`neurons_in_hidden_layers`	a *number* providing the number of *neurons* in each hidden layers.
`datafile`	a *string* providing a path to a binary file in which to save (or load) the network once trained.

By providing a *functor* instead of just a *symbol* for the *terms* in the `generalize` and `formatting` *lists*, *list* and *data terms* can be injested by the *elemental*. For example, if we are expecting 10 inputs to be provided in a *list* or a *data*, we would specify this as `i(10)`. For concrete example, check any of samples in `./etc/samples/ml`.

To dive in the details, have a look at the file `iris.fizz` in the `samples` folder. As the name indicates, this samples uses the famous *Iris dataset* (which you can find in `https://archive.ics.uci.edu/ml/datasets/iris`) which, have been processed into a *fizz Knowledge*. Let's look at how we have set up the *elemental*:

```
iris {  class = FZZCFFBNetwork,
        alias = iris.ffbn,
        query = iris(_,_,_,_,_),
        generalize = [[i,i,i,i,o],[o,i,i,i,i]],
        formatting = [d,d,d,d,l],
        hidden_layers = 1,
        neurons_in_hidden_layers = 4,
} {

}
```

In the example we request the *elemental* object to create two *neural networks* (with the `generalize` label/-value). Both will have four *inputs* and a single *output* neurons, however which of the *terms* is an output is the difference. For the first *network*, we specified `[i,i,i,i,o]` which means the last *term* will be the output. For the second *network*, we have `[o,i,i,i,i]` where the first *term* will be the output. The `formatting` label indicates that the first four *terms* are data while the last *term* is a label.

Unless the *elemental* is already trained, you will need to use the `/tells` console command to instruct the object to collect training data as well as use them to train the networks. Here's an example of this:

```
?- /tells(iris.ffbn,acquires)
?- /tells(iris.ffbn,practice(1.0,1500,0.1))
iris - practice completed (0.000138,0.000000)
iris - practice completed (0.000398,0.000000)
```

Sending the *symbol* `acquires` to the *elemental* will set it into a training data acquisition state in which the `query` you provided in the *properties* (or by using the `/poke` command) will be used to collect *statements*. Depending on how much data can be collected (there's no console feedback) you can wait a little bit before entering the second `/tells` command which instructs the *elemental* to train (*practice*) using the *statements* it has received so far. The parameters provided in the *functor* are (in order): split between training and validation data (a *number* between 0 and 1), the count of *epochs* to train the models for and the learning rate. In this case, we are requesting all received *statements* to be used as training data, the epoch to be `1500` and the learning rate to be `0.1`.

The output on the console for the second `/tells` command will indicate when the training is completed for each *networks*. The numbers in the parantheses are the *training error* and *validation error*. In this case, since we have no validation data, the *validation error* is `0`.

Once the *networks* are trained, the *models* can be used. For example, we can classify:

```
?- #iris(4.40,2.90,1.40,0.20,:x)
-> ( setosa ) := 0.98 (0.001) 1
```

Note the *truth value* for the `iris` statement that was returned by the *elemental* (`0.98`). We can also do a regression to find out a value for the first *term*:

```
?- #iris(:x,2.90,1.40,0.20,setosa)
-> ( 4.838565 ) := 0.99 (0.001) 1
```

Note that having more than one unbound *variable* in your *query* isn't supported. When the *elemental* is saved, the *models* will be saved in the *properties* as a *binary term* under the label `data`.

EV3

The `modEV3` module provides access to the *LEGO Mindstorms EV3*[7] sensors and motors when running *fizz* on the *EV3 Intelligent brick* (it-self running the `Linux` distribution `ev3dev`[8]).

All of the *elementals* provided by the module follows the same patterns when it comes to interacting with them. That is using specific queries to read (peek) values, write (poke) values and execute specific *functions* (call) or cancel running *functions* (halt).

More information on each sensor and motor can be found within the `ev3dev` documentation[9].

[7]https://www.lego.com/en-us/mindstorms/products/mindstorms-ev3-31313
[8]https://www.ev3dev.org/
[9]http://docs.ev3dev.org/projects/lego-linux-drivers/en/ev3dev-stretch/index.html

`EV3CSYSLEGOSystem`

Along with providing a way to read the device's battery status, the *elemental* watches over plugging and unplugging of sensors or motors. It also provides some core functionalities for the other *elementals* in the modules, and as such its presence in the *substrate* is mandatory.

The *elemental* has the following single *property*:

`bat.technology`	because the `EV3` cannot tell what type of the battery powering it, this property provides that information so that the estimation of the battery status can be more accurate. Accepted values are: `liion` and `nimh`.

Several values can be read from the *elemental* using a `peek` *predicate*:

`bat.current`	battery current in *microamps*.
`bat.voltage.min`	nominal battery voltage when *empty* (the value depends on the technology).
`bat.voltage.max`	nominal battery voltage when *full* (the value depends on the technology).
`bat.voltage`	battery voltage in *microvolts*.
`bat.voltage.p`	naive estimation of the battery percentage based on the voltage.

For example, if the *elemental* is labeled `ev3.sys`, we would query any of the above values as follow:

```
?- #ev3.sys(peek,bat.current(:c))
-> ( 188000 ) := 1.00 (0.069) 1
```

Multiple values can be peeked at in the same query by using a *list* of *functor* as the second argument. For example:

```
?- #ev3.sys(peek,[bat.voltage(:v),bat.voltage.p(:p)])
-> ( 7421066 , 0.595722 ) := 1.00 (0.098) 1
```

When sensors or motors are plugged or unplugged from the device ports, the *elemental* will publish *statements* that provide information on such events. For instance:

```
ev3.sys(enum, removed, sensor, 2)
ev3.sys(enum, plugged, sensor, 3)
```

The first one indicate that the sensor identified by the id `2` was removed, and the second one indicates that a sensor was plugged with the id `3`. If the event relate to a motor, the third *term* of the *statement* will be the symbol `motor`.

`EV3CSYSLEGOLed`

The `EV3CSYSLEGOLed` provides a way to control one of the LED available on the *EV3 Intelligent brick*. Each of two LEDs is actually composed of two LEDS, one red and one green. When they are both set to the same value, the LED will be orange.

The *elemental* has the following single *property*:

`index`	indicate which of the LED the *elemental* should access. The value can be `0` for the left LED or `1` for the right one.

The color of the LED can be changed (or queried) using a `peek` or `poke` *predicate*:

`r`	the brightness of the red LED (from 0 to 1).
`g`	the brightness of the green LED (from 0 to 1).

For example, if the *elemental* is labeled `ev3.sys.led.1`, we would change the color of the LED to a not so bright orange as follow:

```
?- #ev3.sys.led.1(poke,[r(0.2),g(0.2)])
-> (  ) := 1.00 (0.028) 1
```

The brightness of the LED can be read as follow:

```
?- #ev3.sys.led.1(peek,r(:b))
-> ( 0.200000 ) := 1.00 (0.046) 1
```

`EV3CACTLEGOMotor`

This *elemental* controls a single LEGO *tacho motor*. The following *properties* can be set at load time but also modified at runtime:

`dutycycle`	the *duty cycle* setpoint of the motor. Accepted range is `-100` to `100`.
`polarity`	polarity of the motor; either the *symbol* `normal` or `inversed`.
`port`	the port on which the motor is connected. Accepted *symbols* are: `portA`, `portB`, `portC` and `portD`.
`rampdw`	ramp down setpoint (in ms).
`rampup`	ramp up setpoint (in ms).
`speed`	target speed in tacho counts per second.
`stopaction`	stop action to be applied at the end of a run (or when the stop command is used). Accepted values are: `coast`, `brake` and `hold`.

Other *properties* can only be set or get at runtime:

`maxspeed`	the maximum speed value for the motor (peek only) in tacho count per second.
`position`	the current position (on peek) or the target position (on poke) in tacho count.
`count`	the number of tacho count in one rotation of the motor.
`state`	the current state of the motor (peek only), as a *list* of any of the following *symbols*: `running`, `ramping`, `holding`, `overloaded`, `stalled`.
`running`	is the motor currently running (peek only) as a boolean value.
`holding`	is the motor currently holding (peek only) as a boolean value.

For example, if the *elemental* is labeled `ev3.act.motor.t`, we would read the position as follow:

```
?- #ev3.act.motor.t(peek,position(:p))
-> ( -44 ) := 1.00 (0.052) 1
```

When it comes to executing *functions*, the *elementals* implements the following:

`by`	runs the motor until its position is offset by a given value.
`for`	runs the motor for a given amount of time (in ms).
`go`	runs the motor until it is stopped.
`reset`	stops the motor and reset the position to 0.
`stop`	stops the motor.
`to`	runs the motor until its position reachs the given value.

When requesting the *elemental* to execute a specify function, the query will be answered right away, even if the *function* has yet to be completed. For example:

```
?- #ev3.act.motor.t(call,by(-45))
-> (  ) := 1.00 (0.037) 1
```

Unlike some of the other *elementals* in the module, the motor one doesn't provide a way to monitor the progress of a *function*. The way to do so, will be to frequently peek at the `position` and/or `state` of the motor.

EV3CSENLEGOColor

This *elemental* provides access to a LEGO *color sensor*, which can be use to sense the reflected or ambient light. The following *properties* can be set at load time but also modified at runtime:

`port`	the port on which the sensor is connected. Accepted *symbols* are: `port1`, `port2`, `port3` and `port4`.
`mode`	the mode of operation of the sensor: `reflected`, `ambient`, `index` or `value`

Depending on the `mode` in which the sensor is set, the reading (via the `value` *property*) from the sensor will be different. The following table details the various supported modes:

`ambient`	ambient light intensity (0 to 1).
`reflected`	reflected light intensity (0 to 1).
`index`	the detected color (any of the *symbols*: `black`, `blue`, `green`, `yellow`, `red`, `white`, `brown`.
`value`	raw color expressed in a `list` of three numbers (red, green, blue).

For example, if the *elemental* is labeled `ev3.sen.color`, we would read the sensor (set in reflected mode) as follow:

```
?- #ev3.sen.color(peek,value(:c))
-> ( 0.010000 ) := 1.00 (0.112) 1
```

EV3CSENLEGOGyros

This *elemental* provides access to the LEGO *gyroscope sensor*, which can be use to sense the direction in which a particular robot is facing. The following *properties* can be set at load time but also modified at runtime:

`port`	the port on which the sensor is connected. Accepted *symbols* are: `port1`, `port2`, `port3` and `port4`.
`mode`	the mode of operation of the sensor (as a *symbol*).
`inverted`	set to `yes` if the sensor is mounted inverted.

The supported modes are:

`angle1axis`	the sensor provides the rotation angle along the first axis (in degrees).
`rrate1axis`	the sensor provides the rotational speed along the first axis (in degrees per second).
`angle2axis`	the sensor provides the rotation angle along the second axis (in degrees).
`rrate2axis`	the sensor provides the rotational speed along the first axis (in degrees per second).

Just like the other sensor based *elementals*, the sensor can be read with a `peek` query. For example, if the *elemental* is labeled `ev3.sen.gyros`, we can get the current value as follow:

```
?- #ev3.sen.gyros(peek,value(:h))
-> ( -46 ) := 1.00 (0.092) 1
```

EV3CSENLEGOPower

The *elemental* EV3CSENLEGOPower provides access to the LEGO *Energy Display* (part of a science kit). The following *properties* can be set at load time but also modified at runtime:

port	the port on which the sensor is connected. Accepted *symbols* are: port1, port2, port3 and port4.

At runtime, the reading from the sensor can be retreive using a peek query. The value is a *list* which contains three *terms*. The first two are *lists* holding the readings (voltage in mili-volts, current in mili-amps and power in mili-watt) respectivelu for the *input* and *output* ports. The last *term* in the list is the energy stored by the device (in Joules).

For example, assuming an *elemental* labeled ev3.sen.power:

```
?- #ev3.sen.power(peek,value(:r))
-> ( [[7.969000, 188, 1485], [9.840000, 0, 0], 48] ) := 1.00 (0.093) 1
```

EV3CSENLEGOSonic

This *elemental* provides access to the LEGO *Ultrasonic Sensor*, which gives an estimation of the distance between the sensor and a possible object. The following *properties* can be set at load time but also modified at runtime:

port	the port on which the sensor is connected. Accepted *symbols* are: port1, port2, port3 and port4.
mode	the mode of operation of the sensor (as a symbole.)

The supported modes are:

continuous	continuous measurement.
occasional	Single measurement.
listening	Listen (for another Ultrasonic sensor)

Independently the mode in which the sensor is set (except listening), the reading (via the value *property*) from the sensor will always be expressed in meters, within the range 0.0 to 2.55. Note that 2.54 is the maximum range of the sensor. When the sensor is in listening mode, the value will be either 0 or 1. The later meaning another device was heard.

Assuming an *elemental* labeled ev3.sen.sonic, we would fetch the latest value from it as follow:

```
?- #ev3.sen.sonic(peek,value(:r))
-> ( 1.227000 ) := 1.00 (0.079) 1
```

EV3CSENLEGOTouch

This *elemental* provides support for the LEGO *Touch Sensor* which can be used to detect contact will objects or act as a button that can be pressed by somebody. The following *properties* can be set at load time but also modified at runtime:

port	the port on which the sensor is connected. Accepted *symbols* are: port1, port2, port3 and port4.

At runtime, the state of the button can be checked by using a *peek* query on pressed as follow:

```
?- #ev3.sen.touch(peek,pressed(:r))
-> ( 0 ) := 1.00 (0.172) 1
?- #ev3.sen.touch(peek,pressed(:r))
-> ( 1 ) := 1.00 (0.098) 1
```

When the button is currently not pressed, the value will be `0`. And it will be `1` when pressed. Whenever the button is pressed or depressed, the *elemental* will publish a *statement* indicating the occurence of such event. The *statement* will be formatted as follow:

```
ev3.sen.touch(hint, pressed(1))
ev3.sen.touch(hint, pressed(0))
```

EV3CBEVDrive

This *elemental* provides a more advanced functionality that combines multiple motors and sensors to perform *Tank steering* driving. The following *properties* can be set at load time but also modified at runtime:

`hints`	control loop frequency (in ms)
`ticks`	how often to publish a hint when driving (modulo on the hints)
`gyros`	label of the gyros sensor *elemental*
`motor.l`	label of the left motor *elemental*
`motor.r`	label of the right motor *elemental*
`odometry`	a *frame* that describes the odometry characteristics to be used. That is: `wheel.c` for the circumference of the wheel (in m), `motor.d` for the measured distance in between the center of the motors (in m)
`move`	a *frame* that describes the setting to be used when the robot is actually driving. That is: `speed` for the speed to be applied to both motors when at full power level. `pid.Kp` for the PID's proportional constant, `pid.Kd` for PID's derivative constant and `pid.Ki` for the PID's integral constant.
`turn`	a *frame* that describes the setting to be used when the robot is turning in place. That is: `speed` for the speed to be applied to both motors when at full power level. `pid.Kp` for the PID's proportional constant, `pid.Kd` for PID's derivative constant and `pid.Ki` for the PID's integral constant.

A few other *properties* can only be set or get at runtime:

`heading`	The target heading (in degree) that should be reached.
`pwlevel`	The power level (as a *number* betweeb `-1` to `1`) to be applied.
`position`	A *list* of two *numbers* giving the position of the robot as estimated by the odometry (X,Y). If poked, the odometry will be reset to the given value.

To get the robot to drive or turn, the *elemental* implements the following *functions*:

`move`	when this *function* executes, the *elemental* will attempt to drive in the direction given by the `heading`. Note that it will not re-orient itself in-place before driving forward. Use the `turn.to` *function* first if that is needed.
`turn.by`	when this *function* executes, the *elemental* will orient the robot to face the current `heading` offsets by a value given as argument to the function.
`turn.to`	when this *function* executes, the *elemental* will orient the robot to face a specific heading, given as argument to the function.

While any of the above *functions* are in progress, the *elemental* will publish *hint statements*. For example, with the `move` *function*, the second *term* of the *statement* will be a *functor* with an arity of three:

```
?- #ev3.bev.drive(poke,pwlevel(0.5)), #ev3.bev.drive(call,move)
spy : Q #ev3.bev.drive(poke, pwlevel(0.500000)) (14.994122)
spy : R ev3.bev.drive(poke, pwlevel(0.500000)) (14.975505)
spy : Q #ev3.bev.drive(call, move) (14.924033)
spy : R ev3.bev.drive(call, move) (14.915030)
-> (  ) := 1.00 (0.149) 1
spy : S ev3.bev.drive(hint, move(-7, [0.017187, -0.002415], 7)) (15.000000)
spy : S ev3.bev.drive(hint, move(-6, [0.049698, -0.006407], 6)) (15.000000)
spy : S ev3.bev.drive(hint, move(-5, [0.084466, -0.010019], 5)) (15.000000)
spy : S ev3.bev.drive(hint, move(-2, [0.136583, -0.014501], 2)) (15.000000)
spy : S ev3.bev.drive(hint, move(-1, [0.151241, -0.014996], 1)) (15.000000)
spy : S ev3.bev.drive(hint, move(1, [0.211121, -0.016041], -1)) (15.000000)
spy : S ev3.bev.drive(hint, move(1, [0.214054, -0.015990], -1)) (15.000000)
spy : S ev3.bev.drive(hint, move(3, [0.263421, -0.015043], -3)) (15.000000)
spy : S ev3.bev.drive(hint, move(4, [0.300769, -0.013077], -4)) (15.000000)
```

The first *term* is the current heading (read from the *gyroscope*), the second *term* is the current position (as estimated by the odometry) and the last term is the error in heading (in degrees).

Once started, the *functions* will keep on running until they are implicitely terminated by commanding the *elemental* with the single *term* `halt`. For example:

```
?- #ev3.bev.drive(halt)
-> (  ) := 1.00 (2.255) 1
```

`EV3CBEVSonar`

This *elemental* combines a single motor, a gyroscope and an Ultrasonic sensor as well as the `EV3CBEVDrive` *elemental* to implement a *sonar* like functionality. The motor is expected to allow for the Ultrasonic sensor to be rotated around the Y (Up) axis. The *elemental* implements two *functions*: `scan`, which supports reading the distance to possible obstacles along a set list of heading offsets from the current orientation of the robot; and `skim` which supports collecting the distance to possible obstacles between two heading offsets in a more continuous way.

The following *properties* can be set at load time but also modified at runtime:

`gyros`	label of the gyros sensor (optional)
`sonic`	label of the ultrasonic sensor
`motor`	label of the motor (that can turn the ultrasonic sensor)
`drive`	label of the drive behavior (optional)
`color`	label of the color sensor (optional)
`scan.mtime`	how often to check if the motor has reached the target position (in ms).
`scan.itime`	how long after a step before reading the sonic sensor (in ms).
`scan.speed`	speed of the motor to be applied in scan mode.
`skim.mtime`	how often to read from the sensor while the motor is turning in skim mode (in ms).
`skim.speed`	speed of the motor to be applied in skim mode.

After each runs of a *function*, the *elemental* will publish a *hint statement* containing the readings that were collected (as well as a timestamp value). Each reading is given as a *list* of three *terms*. For example:

```
?- #ev3.bev.sonar(call,scan([-90,-45,0,45,90]))
spy : Q #ev3.bev.sonar(call, scan([-90, -45, 0, 45, 90])) (14.994852)
spy : R ev3.bev.sonar(call, scan([-90, -45, 0, 45, 90])) (14.978301)
-> (  ) := 1.00 (0.117) 1
spy : S ev3.bev.sonar(hint, scan(1558410099.485098, [[-147, 0.757000, [0, 0]], [-102, 1.904000, [0,
     0]], [-59, 2.550000, [0, 0]], [-15, 2.550000, [0, 0]], [30, 1.002000, [0, 0]]])) (15.000000)
```

The first *term* in the *list* is the heading at which the distance was sample. The second *term* is the distance (in meters) and the third is the position of the robot at the time of the sensing. If the `color` property was provided, the value from the sensor will be added as the fourth *term*.

If the minimum or maximum readings are what is most needed. The *elemental* also support four *functions* that are variation on the two main ones: `scan.min`, `scan.max` and `skim.min`, `skim.max`. Their *hint statements* will still contains the *list* of all readings, but they will also provides the minimum or maximum value (as the last *term* in the *functor*). For example:

```
?- #ev3.bev.sonar(call,scan.max([-90,-45,0,45,90]))
spy : Q #ev3.bev.sonar(call, scan.max([-90, -45, 0, 45, 90])) (14.991117)
spy : R ev3.bev.sonar(call, scan.max([-90, -45, 0, 45, 90])) (14.951246)
-> (  ) := 1.00 (0.166) 1
spy : S ev3.bev.sonar(hint, scan.max(1558410277.380321, [[-147, 2.550000, [0, 0]], [-102, 1.906000,
     [0, 0]], [-59, 2.550000, [0, 0]], [-14, 2.550000, [0, 0]], [30, 1.006000, [0, 0]]], [-147,
    2.550000, [0, 0]])) (15.000000)
```

Unlike with the *drive elemental*, these *functions* will not continuously run. If cyclic execution of any of the *functions* is needed, it can be requested from the *elemental* by providing a time interval as the last *term* in the `call` query. For example:

```
?- #ev3.bev.sonar(call,scan.max([-90,-45,0,45,90],1500))
spy : Q #ev3.bev.sonar(call, scan.max([-90, -45, 0, 45, 90], 1500)) (14.991325)
spy : R ev3.bev.sonar(call, scan.max([-90, -45, 0, 45, 90], 1500)) (14.977512)
-> (  ) := 1.00 (0.146) 1
spy : S ev3.bev.sonar(hint, scan.max(1558410741.919013, [[-146, 1.301000, [0, 0]], [-102, 1.906000,
     [0, 0]], [-59, 2.550000, [0, 0]], [-14, 2.550000, [0, 0]], [30, 1.007000, [0, 0]]], [-59,
    2.550000, [0, 0]])) (15.000000)
spy : S ev3.bev.sonar(hint, scan.max(1558410746.059581, [[-147, 1.305000, [0, 0]], [-102, 1.913000,
     [0, 0]], [-59, 2.550000, [0, 0]], [-14, 2.550000, [0, 0]], [30, 1.006000, [0, 0]]], [-59,
    2.550000, [0, 0]])) (15.000000)
spy : S ev3.bev.sonar(hint, scan.max(1558410749.724064, [[-147, 1.305000, [0, 0]], [-102, 1.904000,
     [0, 0]], [-59, 2.550000, [0, 0]], [-14, 2.550000, [0, 0]], [30, 1.006000, [0, 0]]], [-59,
    2.550000, [0, 0]])) (15.000000)
spy : S ev3.bev.sonar(hint, scan.max(1558410753.130281, [[-147, 1.302000, [0, 0]], [-103, 1.913000,
     [0, 0]], [-58, 2.550000, [0, 0]], [-13, 2.550000, [0, 0]], [30, 1.007000, [0, 0]]], [-58,
    2.550000, [0, 0]])) (15.000000)
```

To get the current reading from the *sonar*, you can *peek* at the *value* of it like this:

```
?- #ev3.bev.sonar(peek,value(:v))
-> ( [30, 0.471000, [0, 0], 0.010000] ) := 1.00 (0.172) 1
```

`EV3CSRVMapping`

This *elemental* supports building a localization map using a range finder sensor and using that map to compute a path between obstacles. When used with the Ultrasonic sensor, the ability to localize the robot is unusable due to the large field-of-view of the sensor. Note that this *elemental* is only available on the `x64` build of the module.

The following *properties* can be set at load time to setup the mapping:

`resolution`	size of a voxel (in m)
`width`	width of the map (in m)
`height`	height of the map (in m)
`range`	effective range of the sensor (in m)
`decay`	how long an obstacle stays in the map (in s)
`decay.tick`	how often to apply decay to the map (in ms), default is 0

and the following *properties* can be used to setup the path finding settings:

`iweight`	inflation weight during path finding (the bigger the number the longer the algorithm will take ...)
`inflate`	default inflation range (a list of two numbers) around every occupied voxel (in m) of the map
`dscales`	distance scale(s) used by the distance function (a list of two numbers)
`dfunct`	distance function to be used: **`Euclidean`**, **`Manhattan`** or **`Diagonal`**
`direct`	set to **`yes`** to use direct path whenever possible

The *elemental* implements a collection of *functions* which can be called (via a *predicate*) to update the map or compute a path:

(**`update`**,:p,:h,:a,:d)	takes a scan composed of relative angles (`d`), distances (`a`) from a sensor along with the position (`p`) and heading (`h`) of the sensor and update the map.
(**`localize`**,{},:p,:h,:a,:d,:P,:H,:S)	takes a scan composed of relative angles (`d`), distances (`a`) from a sensor along with the position (`p`) and heading (`h`) of the sensor and attempt the compute the actual pose of the sensor (position (`P`), heading (`H`)) and score (`S`).
(**`reset`**)	reset the map.
(**`load`**,*string*)	load a previously saved map from a PPM file.
(**`save`**,*string*)	save the current map into a PPM file.
(**`plan`**,:s,:t,:l)	compute a path in between two points (`s` to `t`) and unifies the list of points with the last term (`l`).
(**`near`**,:p,:d,:P)	find a point on the map (`P`) which is not occupied and within a given distance (`d`) of the provided point (`p`)
(**`cast`**,:s,:t,:d,:l)	raycast a circle of a given radius (`d`)from a point (`s`) to another (`t`) and return any contact in a *list* unified with the last *term* (`l`).

SQL

The `modSQL` module provides an interface for using *SQLite3*[10] files.

`SQLCDatabase`

This *elemental* manage a single database and provide an interface to searching, updating or managing it. Storing *terms* other than *strings*, *symbols* or *numbers* is supported only if the column type is definied as *blob*. It supports the following *properties*:

`dbfile`	the path to the SQLite file.
`threads`	the number of dedicated threads to be used to answer queries against the *elemental*.
`slimit`	the maximum number of SQL statements that will be cached at the same time.
`schema`	if the SQLLite file doesn't exist, the **`frame`** set will be used to create the tables, just as the **`create`** function would have done it.
`symbols`	a frame indicating for each table which of the column's content is a **`symbol`** (instead of a **`string`**).

[10] https://sqlite.org/

The *elemental* implements a collection of *functions* which can be called (via a *predicate*):

(`create`,:f)	creates (one or more) new tables using a frame given as the second term as the *schema*.
(`delete`,...)	deletes any rows from given tables that matchs.
(`inject`,:o,:s,:b,:r)	executes an SQL statement provided as a string (`s`) and a *list* of values to be bound to the `?` presents in the string with a set of options given in a *frame* (`o`) and unifies all returned rows with the last term (`r`).
(`insert`,:s,...,:r)	inserts a series of rows in the corresponding tables. Each row is given as a *functor* with a single term: a *frame* or a *list*. The last *term* (`r`) will unify with a list of the *rowid* for each of the inserted row.
(`remove`,:l)	removes from the database all the tables whoes name is given in a list (`l`). When no second *term* is given, all the tables will be removed.
(`schema`,:s)	extracts the *schema* of the database and unifies it (as a *frame*) with the second *term*.
(`search`,:o,...)	searches the database by matching *functors* against tables and rows with a set of options given in a *frame* (`o`).
(`update`,...)	updates a series of rows in the corresponding tables. Each row is given as a *functor* with two *terms*: a first *frame* specifying which rows to match and a second *frame* providing which columns to update.
(`gather`,:o,:f,:l)	gather all rows that matches `f` in a list bound to the variable `l`.

Let's look at an example (`etc/samples/sql/cars.fizz`) where we create a new database and import data into it from a `CSV` file. We first defines the database *elemental* as follow:

```
db {

    class   = SQLCDatabase,
    dbfile  = "./cars.db",
    verbose = yes

} {}
```

When the *elemental* is attached to the *substrate*, the database will be opened and stay as such until the *elemental* is detached. If the database didn't exists, we need to create a table in it. For that purpose, we are going to define another *elemental* to hold some *procedural knowledge* that we can use to perform simple management tasks, such as creating a table:

```
db.admin {

    schema = {
                cars =  {
                        maker           = {type = TEXT},
                        model           = {type = TEXT},
                        options         = {type = BLOB},
                        mpg             = {type = REAL},
                        cylinders       = {type = INT},
                        displacement    = {type = INT},
                        horsepower      = {type = REAL},
                        weight          = {type = INT},
                        acceleration    = {type = REAL},
                        year            = {type = INT},
```

```
15                          origin          = {type = TEXT}
16                          }
17     }
18
19 } {
20
21     (create)    :-  #db(remove),
22                     #db(create,$schema);
23
24 }
```

We use the *property* `schema` in the *elemental* `db.admin` to defines the *layout* of the table we want to have in the database. Here, we defines a table named `cars` with 11 columns. Each of the columns is specified via a *frame*. The supported *properties* of a columns are:

`type`	the data type (as a *symbol*) of the column as defined by SQLite3 (e.g. `TEXT`, `BLOB`, `REAL`, `INT`, `NUMERIC` ...). The value can also be a *functor* for example, to specify `VARCHAR(255)`.
`default`	the default value for the column.
`pk`	set to `yes` if the column is the *primary key*.
`fk`	set to `yes` if the column is a *foreign key*.
`notnull`	set to `yes` if the column cannot be null.
`references`	used along `fk`, it is a *functor* that describs the table and column to which the key point to.

The definition contains handling for a single query with the *term* `create`, which calls the database *elemental*, to first remove all existing tables (line 21) before using a `create` based query to create the `cars` table.

To import some data into the database, we are not going to use a CSV file (`etc/data/cars.csv`) whoes content is the following:

```
Chevrolet Chevelle Malibu;18.0;8;307.0;130.0;3504.;12.0;70;US
Buick Skylark 320;15.0;8;350.0;165.0;3693.;11.5;70;US
Plymouth Satellite;18.0;8;318.0;150.0;3436.;11.0;70;US
AMC Rebel SST;16.0;8;304.0;150.0;3433.;12.0;70;US
...
```

The columns in that file more or less match the *schema* we have setup, except for the first column which we will split into three columns: `maker`, `model` and `options` based on the observation that it is devided as such. For that, we defines the following two *elementals*:

```
db.convert { // convert the statement published by the CSV importer into data to be inserted

    ()  :-  @car.import(:str,:mpg,:cyl,:dis,:hor,:wei,:acc,:yea,:ori),
            str.tokenize(:str," ",:lst), #db.proc.name(:lst,:maker,[:model|:options]),
            #db.admin(insert,[:maker,:model,:options,:mpg,:cyl,:dis,:hor,:wei,:acc,:yea,:ori]);

}

db.proc.name { // process the car name to separate the maker/model

    ([],"","")              ^:- true;
    ([:maker],:maker,:maker) ^:- true;
    (:l,:maker,:model)       ^:- lst.head(:l,:maker), lst.rest(:l,:model);
```

```
14
15 }
```

In the first one, the publication of a *statement* labeled `car.import` (line 3) will trigger the insertion of a new row in the table after some processing of the maker, model and options (line 5). We can now modify the definition of `db.admin` to add support for starting the import of the CSV file as well as the insertion of each row into the database:

```
db.admin {

    schema = {
                cars =  {
                        maker           = {type = TEXT},
                        model           = {type = TEXT},
                        options         = {type = BLOB},
                        mpg             = {type = REAL},
                        cylinders       = {type = INT},
                        displacement    = {type = INT},
                        horsepower      = {type = REAL},
                        weight          = {type = INT},
                        acceleration    = {type = REAL},
                        year            = {type = INT},
                        origin          = {type = TEXT}
                        }
    }

} {

    (create)    :-  #db(remove),
                    #db(create,$schema);

    (import)    :-  console.exec(import.csv("./etc/data/cars.csv",car.import,";",[],2));

    (insert,:cols?[is.list])    :-  lst.length(:cols,11),      // verify the number of columns
                                    #db(insert,cars(:cols),_); // insert the new "row"

}
```

Let's now try this:

```
$ ./fizz.x64 ./etc/samples/sql/cars.json
fizz 0.7.0-X (20200710.1423) [lnx.x64|8|l]
Press the ESC key at anytime for input prompt

load : loading ./etc/samples/sql/cars.json ...
load : loaded ./mod/lnx/x64/modSQL.so in 0.000s
load : loading ./etc/samples/sql/cars.fizz ...
db - database opened
load : loaded ./etc/samples/sql/cars.fizz in 0.012s
load : loading completed in 0.014s
?- #db.admin(create)
-> (  ) := 1.00 (0.045) 1
?- #db.admin(import)
-> (  ) := 1.00 (0.001) 1
import.csv : 406 lines read in 0.017s.
```

Since the CSV file import is asynchronous, there's no direct way to know when it has completed, but after a couple of seconds, we can assume it has completed, so let's count the number of rows in the table we should have 404 cars in the database (the first two lines of the CSV file are to be ignored):

```
?- #db(inject,{},"SELECT count(model) FROM cars;",[],:l)
-> ( [unknown({count(model) = 404})] ) := 1.00 (0.001) 1
```

Alright, that's correct. Let's look at the first car in the database:

```
?- #db(inject,{},"SELECT * FROM cars LIMIT 1;",[],:l)
-> ( [cars({maker = "Chevrolet", model = "Chevelle", options = ["Malibu"], mpg = 18, cylinders = 8,
     displacement = 307, horsepower = 130, weight = 3504, acceleration = 12, year = 70, origin = "
    US"})] ) := 1.00 (0.005) 1
```

Instead of providing a SQL statement, we could have for this simple case, use the `search` function of the *elemental* as follow:

```
?- #db(search,{limit=1},cars(:f))
-> ( {maker = "Chevrolet", model = "Chevelle", options = ["Malibu"], mpg = 18, cylinders = 8,
    displacement = 307, horsepower = 130, weight = 3504, acceleration = 12, year = 70, origin = "US
    "} ) := 1.00 (0.001) 1
```

We can use this function to ask the database questions such as: *which Dodge got released in 1974?*:

```
?- #db(search,{},cars({model = :m, maker = "Dodge", year = 74}))
-> ( "Coronet" ) := 1.00 (0.001) 1
-> ( "Colt" ) := 1.00 (0.001) 2
```

Let's reformulate this query using a SQL statement and bindings:

```
?- #db(inject,{},"SELECT model FROM cars WHERE maker = ? AND year = ?;",["Dodge",74],:l)
-> ( [cars({model = "Coronet"})] ) := 1.00 (0.001) 1
-> ( [cars({model = "Colt"})] ) := 1.00 (0.001) 2
```

Bindings allow us to wrap a complex SQL statement inside of a *procedural knowledge* such as this:

```
released { // what cars from a given maker got released in a given year

    (:m,:y,:M) :- #db(inject,{},"SELECT model FROM cars WHERE maker = ? AND year = ?;",
                      [:m,:y],[cars({model = :M})]);

}
```

We can then query it like we would do for any other *elemental*:

```
?- #released("Dodge",75,:l)
?- #released("Dodge",74,:l)
-> ( "Coronet" ) := 1.00 (0.002) 1
-> ( "Colt" ) := 1.00 (0.002) 2
?- #released("Dodge",73,:l)
```

```
-> ( "Coronet" ) := 1.00 (0.002) 1
-> ( "Dart" ) := 1.00 (0.002) 2
```

Now, let's try to answer the question "which car released in 1974 has the lowest horsepower?":

```
?- #db(inject,{},"SELECT maker, model, horsepower FROM cars WHERE year = 74 ORDER BY horsepower ASC
     LIMIT 1;",[],[cars({maker=:k,model=:m,horsepower =:h})])
-> ( "Ford" , "Maverick" , 0 ) := 1.00 (0.001) 1
```

`0`? That can't be right. Is there more?

```
?- #db(search,{},cars({rowid = :r, maker = :k, model = :m, horsepower = 0, year = :y}))
-> ( 39 , "Ford" , "Pinto" , 71 ) := 1.00 (0.001) 1
-> ( 134 , "Ford" , "Maverick" , 74 ) := 1.00 (0.002) 2
-> ( 337 , "Renault" , "Lecar" , 80 ) := 1.00 (0.002) 3
-> ( 343 , "Ford" , "Mustang" , 80 ) := 1.00 (0.002) 4
-> ( 360 , "Renault" , "18i" , 81 ) := 1.00 (0.002) 5
-> ( 381 , "AMC" , "Concord" , 82 ) := 1.00 (0.002) 6
```

Turns out, the CSV file have errors which we will correct using the `update` function of the *elemental* as follow (using more correct data from Wikipedia):

```
?- #db(update,cars({rowid = 39},{horsepower = 75}))
-> (  ) := 1.00 (0.016) 1
?- #db(update,cars({rowid = 134},{horsepower = 75}))
-> (  ) := 1.00 (0.023) 1
?- #db(update,cars({rowid = 337},{horsepower = 55}))
-> (  ) := 1.00 (0.022) 1
?- #db(update,cars({rowid = 343},{horsepower = 80}))
-> (  ) := 1.00 (0.016) 1
?- #db(update,cars({rowid = 360},{horsepower = 63}))
-> (  ) := 1.00 (0.014) 1
?- #db(update,cars({rowid = 381},{horsepower = 82}))
-> (  ) := 1.00 (0.013) 1
```

We can verify the update as follow:

```
?- #db(search,{},cars({rowid = 337, model = :m, horsepower = :h, year = :y}))
-> ( "Lecar" , 55 , 80 ) := 1.00 (0.001) 1
```

STR

The `modSTR` module supports the connection of multiple instances of *fizz* running on multiple computers. Unlike the `modCLU`, this relies on the TCP protocol and as such a single instance is the *server* and the other instances are *clients*.

`STRCGateway`

`STRCGateway` is the *elemental* which provides a link between the local instance of *fizz* and the other instances. For this to work, each of the *client* instances must know the IP (or name) of the server and the port number. Both can be specified using the properties of the *elemental* object:

`mode`	specify either the *elemental* is a `server` or a *client*.
`host`	if the *elemental* is a *client*, this specify the name or IP of the *server* to connect to (using a *string* or *symbol*). The default value is `localhost`.
`port`	the port number used by the *server*.
`filters`	a list of the *statements* and *predicates* labels to be accepted for incoming and outgoing transmissions

Any traffic coming from a *client* will be forwarded to all other *clients*, therefore, each instance only need to connect to the *server* to participate in any logical inference.

PYT

The `modPYT` module supports executing *Python* (3.6.X) code from within *fizz* .

`PYTCModule`

`PYTCModule` is the *elemental* which provides the interface between a *Python* module and the *substrate*. The following properties of the *elemental* object are available:

`path`	the folder in which the python file may be located.
`name`	the name of the python module to be loaded.
`object`	an optional *symbol* or *functor* which will be interpreted as the class of object to be instantiated. When an object is instantiated, every *query* will to the *elemental* will be transformed into a method call to that object.

In order to call a function defined in the indicated *Python* module, a *query* to the *elemental* must contains a *functor* that represents the call. For example. If we had function defined in a *Python* file called `leap.py`, which given the current year, returns a list of all the leap years over the next 10 years:

```
import calendar
from datetime import datetime

def next():
    y = datetime.today().year
    l = []
    for i in range(y, y+10):
        if calendar.isleap(i):
            l.append(i)
    return l
```

We would setup the *elemental* as follow, assuming the *Python* file is located in the current folder:

```
py.leap {
    class   = PYTCModule,
    path    = "./",
    name    = "leap"
}
```

Once the *elemental* is loaded (remember to load the module `modPYT` before), we can query it as such:

```
?- #py.leap(next(),:l)
-> ( [2020, 2024, 2028] ) := 1.00 (0.001) 1
```

As shown, the *query* first *term* is marshalled into a *Python* function call. The result of the call on the *Python* side, will be marshalled back into a *fizz term* and unified to the second *term* of the *query*.

Since not all the *term* in *fizz* have a corresponding *Python* type, the *module* defines a few custom *Python* class:

`fizz.Data`	marshalls the *data* term. The property `bytes` provides access to a `bytearry` *Python* object. The property `format` provides the format of the data as an unsigned integer code.
`fizz.Func`	marshalls the *func* term. The property `label` provides the label of the *functor* as a string and the property `terms` provides the *terms* of the *functor* as a tuple.
`fizz.Quirk`	marshalls to *quirk* term, with the properties `head` and `tail` providing access to the content of the *quirk*.
`fizz.Range`	marshalls to *range* term, with the properties `min` and `max` providing access to values (as numbers).
`fizz.Variable`	marshalls the *variable* term. The property `label` provides the label of it, and the property `constraint` providing any set constraint.

For a data's format, the mapping is as follow:

`Byte`	1
`Char`	2
`Bool`	3
`Uint16`	4
`Sint16`	5
`Uint32`	6
`Sint32`	7
`Uint64`	8
`Sint64`	9
`Real32`	10
`Real64`	11

`PYTCElemental`

`PYTCElemental` is an *elemental* which is bound to an instance of a *Python* class. It supports implementing a (limited) *elemental* in *Python* The following properties of the *elemental* object are available:

`path`	the folder in which the python file may be located.
`name`	the name of the python module to be loaded.
`object`	a *symbol* or *functor* which will be interpreted as the class of object to be instantiated.

For this to work, the *Python* class must derive from the class `fizz.Elemental` (You will need to `import fizz` in the *Python* file) where the class definition will be. The `fizz.Elemental` class defines the following methods which can be used to handle specific events:

`onAttach(self)`	called when attached to the *substrate*.
`onDetach(self)`	called when detached from the *substrate*.
`onQuery(self,query)`	called when a *query* is received by the *elemental*. If the call returns a list, it will be interpreted as a list of *statements* to be sent as replies.
`onReply(self,reply,statements)`	called when a *reply* was received by the *elemental* for a *query* it put out.
`onPulse(self)`	called regurlarly.
`onSquib(self,statement)`	called when a *statement* was published by another *elemental*
`onScrap(self,context)`	called when a previously received *query* was discarded.

The class also provides the following methods:

`reply(query,statements)`	provides a list of *statements* to be provided as *reply* to a received *query*.
`query(label,terms)`	send a *query* out based on its label and a list of terms.
`disclose(labels)`	request the *elemental* to pay attention to any *statements* published given one (or more) labels.
`withhold(labels)`	request the *elemental* to stop paying attention to any *statements* published given one (or more) labels.

For the above methods and callbacks, the `query` argument contains a *Python* dictionary which can be used to interpret the query. For example:

```
{'context': [UUID('874fb58b-c341-2f4f-8c17-388db4dead53'), 0, 0, 0], 'ttl': 4.999900817871094,
'stp': 1608401251.001843, 'terms': [<Variable object at 0x7f10e4187bd0>],
'limit': <Range object at 0x7f10e4099210>}
```

The `context` is a list which unquely identify the query. When replying to a query using the method `reply`, this value will have to be passed to the method call. The value `terms` holds the list of *terms* that were used in the query. This is the list that will need to be parsed to produce a reply that fits the purpose of the *elemental*. The value `limit` holds a `fizz.Range` object which provides the accepted *truth value* range.

For `onReply`, the `reply` argument will also contains a *Python* dictionary which is pretty similar to the one above:

```
{'context': [UUID('af26695b-bc38-4d4c-90ce-6ea8c5fa3cd2'), 0, 0, 0], 'label': 'a',
'ttl': 4.999590873718262, 'stp': 1608401728.633754}
```

The `label` and `context` values can be used to disambiguate which of the query the reply is for. The `statements` argument will hold a list of the *statements* that were received as reply. Each of the *statement* being represented in *Python* as a list of size two: a list of *terms* and the truth value of the statement.

In the case of the `onScrap` callback, the `context` argument will hold a list such as the one seen earlier identifying the received query that is being scrapped.

To illustrate this, here is a simple example of a *Python* class which count the number of times the user have pressed a given key using the published *statement*:

```
import fizz

class Counter(fizz.Elemental):

    def __init__(self,key):
        self.key = key
        self.n = 0

    def onAttach(self):
        self.disclose('console.keypress')

    def onDetach(self):
        self.withhold('console.keypress')

    def onQuery(self,query):
        return [([self.n],1)]

```

```
    def onSquib(self,statement):
        if statement[0] == 'console.keypress':
            if statement[1][0] == self.key:
                self.n = self.n + 1
```

To create an *elemental* object mapping to it in the *runtime*, we would declare it as follow:

```
key {
    class   = PYTCElemental,
    path    = $self.path,
    name    = "dcounter",
    object  = Counter(100)
}
```

After loading the *fizz* file and pressing the `d` key a few time, we would query it like any other *elemental* object:

```
?- #key(:x)
-> ( 4 ) := 1.00 (0.001) 1
```

8 Advanced topics

Miscellaneous

`Escaper`

An *Escaper* is a special kind of *term* which utility comes to light, mainly, when used with *volatiles*. It provides a way to protect a *term* from an upcoming *substitution*. As example, let's look at using the `define` *primitive* to create a *prototype* which will provide a function similar to the `assert` *primitive* but with the difference that we will stamp the created *statements*. If we were to create that in a text editor we would do something like this:

```
assert.stamp {

    (:f, :v) :- assert(:f, :v, {stamp = %now});

}
```

To create it from the console, we would type this:

```
?- define(assert.stamp,[\:f,\:v],[],[[[primitive],assert(\:f,\:v,{stamp = \%now})]])
-> (  ) := 1.00 (0.000) 1
```

In it, we have use \ to indicate each of the *terms* which need to be escaped. This will prevent the *volatile* `now` from being substituted when the `define` primitive is called. For convenience, we have also escaped the *variables* `:f` and `:t`. This will prevent the console from expecting the call to `define` to bound the *variables*. Once *escaped* a *term* will stay that way until it is unescaped using the *primitive* `nab`. The *primitive* `define` which we are using in this example will un-escape all *terms*.

We can now test the new `assert.stamp` *prototype* and verify that each of the *statements* is created with a timestamp in its *properties*:

```
?- #assert.stamp(hello(bob),1)
-> (  ) := 1.00 (0.001) 1
?- #assert.stamp(hello(alice),1)
-> (  ) := 1.00 (0.001) 1
?- #hello(:x) {stamp = :s}
-> ( bob , 1509431500.377723 ) := 1.00 (0.001) 1
-> ( alice , 1509431507.226000 ) := 1.00 (0.001) 2
```

Have we not escaped the `now` *volatile*, it will have been substitued during the `define` call and each of the *statements* we would have created will have had the same value for timestamp:

```
?- define(assert.stamp,[\:f,\:v],[],[[[primitive],assert(\:f,\:v,{stamp = %now})]])
-> (  ) := 1.00 (0.000) 1
?- #assert.stamp(hello(bob),1)
-> (  ) := 1.00 (0.001) 1
?- #assert.stamp(hello(alice),1)
-> (  ) := 1.00 (0.001) 1
?- #hello(:x) {stamp = :s}
-> ( bob , 1509433383.169334 ) := 1.00 (0.001) 1
-> ( alice , 1509433383.169334 ) := 1.00 (0.001) 2
```

Lastly, the *runtime environment* defines a primitive called `is.escaper` which can be used to test if a *term* is an *escaper* or not. To force such *term* to surrender the *term* it is protecting, you can use the *primitive* `nab` to bind the *escaped term* to a *variable*.

Services

This section provides some details on all the *services* supported by the *runtime*.

MRKCCollector

The `MRKCCollector` *service* provides a way to assemble all the *statements* generated by a *predicate* and provide them as *lists*. It can be used by use of the `fzz.collect` *predicate*:

`fzz.collect(`*list*,*functor*,*list*|*variable*,*frame?*`)`

The first *term* is a *list* which can contains *symbol* and/or a *range*. Its purpose is to indicate if the *predicate* to collect is negated (`negate` *symbol*) and/or a primitive (`primitive` *symbol*). When a *range* is expressed in the *list*, it will be used as the *predicate* truth value range. The second *term* is a *functor* which express the *predicate* to be collected. Each of the unbound *variables* that will be used in the *functor* will be considered as a target for collection. The third *term* will unify or substitue with a *list* containing the *truth value* of all received *statements*. If provided, the fourth *term* is a *frame* which can specify a timeout value (in seconds) after which the collection will be terminated (with the label `tmo`) if no more *statements* are being collected. When no timeout is provided, the default is half a second. Instead of using a numerical value, the timeout can be specified using the following three *symbols*: `min`, `max` and `avg`. In such cases, the service will estimate a timeout value based on the minimum, maximum or average of the elapsed time between replies. In such case, a *functor* can be used to provide a modifier value to be applied to the computed value, for example `max(1.5)` will multiply the average by `1.5` before comparing it. In any case, the service will only returns what was collected once the timeout occurs.

As an example, let's consider the following knowledges:

```
product {

    (model_e,tesla,2012);
    (iphone_x,apple,2018);
    (vive,htc,2015);
    (coconut_water,zico,2000);

}

product {

    (iphone,apple,2007);
    (iphone_3GS,apple,2009);
    (7710,nokia,2005) := 0.9;

}
```

If we wanted to get the name and year of release of all products with a truth value above `0.9`, we would query:

```
?- #product(:label,_,:years) <0.91|1>
-> ( model_e , 2012 ) := 1.00 (0.001) 1
-> ( iphone_x , 2018 ) := 1.00 (0.001) 2
-> ( vive , 2015 ) := 1.00 (0.001) 3
-> ( coconut_water , 2000 ) := 1.00 (0.001) 4
-> ( iphone , 2007 ) := 1.00 (0.001) 5
-> ( iphone_3GS , 2009 ) := 1.00 (0.001) 6
```

Now, to generate *lists* from the *statements* of all the possible values of the *variables*, we would kick the *predicate* to the service and chain the call like any other *predicate* dealing with *knowledge*:

```
?- #fzz.collect([<0.91|1>],product(:values,_,:years),_), lst.length(:values,:length)
-> ( [iphone, iphone_3GS, model_e, iphone_x, vive, coconut_water] ,
     [2007, 2009, 2012, 2018, 2015, 2000] , 6 ) := 1.00 (0.488) 1
```

`MRKCEvently`

The `MRKCEvently` *service* provides a way to synchronize two (or more) inference execution by providing a mean to wait for an event or signal an event. It can be used by using a `fzz.evently` *predicate*:

`fzz.evently(await,`*atom*`,`*variable*`|`*term*`,`*frame?*`)`
`fzz.evently(flash,`*atom*`,`*term*`)`
`fzz.evently(sleep,`*number*`)`

When the first *term* is the *symbol* `await`, the *predicate* will wait for an event identified by the second *term* and unify the event's value with the third *term*. By providing a *frame* as the fourth *term*, the wait can be setup with a timeout value (`tmo`, expressed in seconds) an/or requested to support awaiting for more than one flashing of the event (set `multi` to `yes`).

When the first *term* is `flash`, the *predicate* will signal any other inference waiting and provide them with the value provided as the third *term*. If instead the first *term* is `sleep`, the service provides a way for an *elemental* to wait for given amount of seconds.

For a concrete example, check the sample file `etc/samples/db/tools.fizz`.

`MRKCEvaluator`

The `MRKCEvaluator` *service* provides a way to evaluate a *functor* like if it was a *predicate*. It can be used by using a `fzz.eval` *predicate*:

`fzz.eval`(*list*,*functor*|*list*,*frame?*)

The first *term* is a *list* which can contains *symbol* and/or a *range*. Its purpose is to indicate if the *predicate* to collect is negated (`negate` *symbol*) and/or a primitive (`primitive` *symbol*). When a *range* is expressed in the *list*, it will be used as the *predicate* truth value range. The second *term* is a *functor* or a *list* which express the *predicate* to be evaluated. If provided, the third *term* is a *frame* which can specify a timeout value (in seconds) after which the evaluation will be terminated (with the label `tmo`). When no timeout is provided, the default will be the *substrate's* (or the *elemental's*) Time-to-live value (`ttl`).

If we look at the previous example, we could have used it as follow:

```
?- #fzz.eval([],product(:name,apple,_),{tmo=2})
-> ( iphone_x ) := 1.00 (2.029) 1
-> ( iphone ) := 1.00 (2.029) 2
-> ( iphone_3GS ) := 1.00 (2.029) 3
```

This service can get more interesting when combined with the use of `fun.make` (see Section 5.7 on page 66) to create the *functor* to be evaluated:

```
?- fun.make(product,[:name,apple,_],:func), #fzz.eval([],:func)
-> ( iphone , product(iphone, apple, 2007) ) := 1.00 (0.733) 1
-> ( iphone_3GS , product(iphone_3GS, apple, 2009) ) := 1.00 (0.733) 2
-> ( iphone_x , product(iphone_x, apple, 2018) ) := 1.00 (0.733) 3
```

Release notes

0.8.6-X

Changes

- elementals:
  - `SQLCDatabase` supports the property `schema` (see section 7 on page 138)
  - `SQLCDatabase` supports foreign key and references (see section 7 on page 138)
  - new property `url.escape` for `FZZCWebAPIGetter` (see section 7 on page 125)
  - `MRKCCCollector` supports `timeout` modes (see section 8 on page 148)
- primitives:
  - `sym.cat` behavior with *list terms* (see section 5.16 on page 93)
  - `sym.tokenize` will convert numbers into *numbers*.
  - `is.bound` accepts more than one *term*.

Additions

- elementals:
  - *class* `FZZCWebURLGetter` (see section 7 on page 128)
- terms:
  - `neq.nor.in` *variable constraint*
  - `frm.label` *lambda*
  - `now`, `lst.empty` as *primitives* for `FZZCFUNRunner`.
- primitives:
  - `lst.bulk` (see section 5.8 on page 68)
- volatiles ((see section 3.10 on page 18):
  - `seq`
  - `sym.16`
  - `uid`

Bug Fixes

- Incorrect handling of double quote and single quite characters in `SQLCDatabase`
- Incorrect error code returned when an error occured in generating SQL statement in `SQLCDatabase`
- Padding wasn't zero-ed when flattening terms leading to possible non-match of terms stored as blob in `SQLCDatabase`
- *Boolean* properties for `Elemental` didn't accept symbols (such as `yes`, `false` ...)
- Crash when unifying a `functor` which label is a *variable*.
- primitive `fun.make` wasn't unifying correctly when its third *term* was not a final *term*.
- Issue with early fail reply when using cascade mode in `MRKCBFSolver`.
- Issue with `fzz.collect` not replying under some condition.
- `sym.tokenize` and `str.tokenize` did not accepts *split-list* as 3rd *term*.
- Issue with inference stopping seemingly randomly.

0.8.0-X

Changes

- elementals:
  - MRKCCSVStore can store *statements* (see section 6 on page 113)
  - FZZCFUNRunner support for primitive `frm.labels` (see section 6 on page 116)
  - FZZCFUNRunner support for primitive `cls` (see section 6 on page 116)
  - FZZCFUNRunner support for primitive `unify` (see section 6 on page 116)
  - FZZCFUNRunner support for primitive `inquire` (see section 6 on page 116)
  - FZZCFUNRunner support for primitive `lst.append` (see section 6 on page 116)
  - FZZCFUNRunner support for primitive `lst.prepend` (see section 6 on page 116)
  - FZZCWebPoster: support for `multipart/form-data` content (see section 7 on page 126)
  - MRKCLettered: support for property `loose` (see section 6 on page 122)
- primitives:
  - `console.gets` accepts two terms, the first one being a prompt to be printed
- samples:
- terms:
  - `if` variable constraint
  - `is.final` variable constraint
  - *frame* can be used as variable constraint
- console:
  - Solution can file can be made to load source files sequentially (see section 4.3 on page 26)
  - A custom `ttl` value can be specified for a *predicate*

Additions

- samples:
  - cnet
  - sql
  - funx4.fizz
  - sentim.fizz
  - pyt
- elementals:
  - PYTCElemental (see section 7 on page 145)
  - PYTCModule (see section 7 on page 144)
  - MRKCMingler (see section 6 on page 112)
  - MRKCCatcher (see section 6 on page 112)
  - SQLCDatabase (see section 7 on page 138)
  - FZZCWebAPIPoster (see section 7 on page 126)
  - MRKCAggregator (see section 6 on page 111)
  - MRKCProxy (see section 6 on page 114)
- terms:
  - `lambda` (see section 3.12 on page 19)
  - `split frame` (see section 3.4 on page 13)

- `eq.or.in` *variable constraint*

- constants:
  - `self.path` (see section 3.9 on page 18)
- primitives:
  - `fzz.stats` (see section 5.11 on page 84)
  - `fzz.parse` (see section 5.11 on page 84)
  - `fzz.exists` (see section 5.11 on page 83)
  - `sleep` (see section 5.2 on page 54)
  - `console.quit` (see section 5.2 on page 48)
  - `rnd.list` (see section 5.13 on page 87)
  - `lst.permu` (see section 5.8 on page 74)
  - `frm.there` (see section 5.6 on page 66)
  - `frm.same` (see section 5.6 on page 66)
  - `var.make` (see section 5.11 on page 86)
  - `sym.end` (see section 5.16 on page 94)
  - `sym.cut` (see section 5.16 on page 94)
  - `bin.length` (see section 5.4 on page 59)
  - `bin.load` (see section 5.4 on page 59)
  - `bin.save` (see section 5.4 on page 59)
  - `sym.tolower` (see section 5.16 on page 95)
  - `sym.toupper` (see section 5.16 on page 96)
  - `var.tofu` (see section 5.11 on page 86)
  - `var.defu` (see section 5.11 on page 85)
  - `hash` (see section 5.2 on page 50)
  - `is.bound` (see section 5.18 on page 102)
  - `sym.tokenize` (see section 5.16 on page 95)
  - `lst.flat` (see section 5.8 on page 70)
  - `str.dist` (see section 5.17 on page 96)
  - `sym.stem` (see section 5.16 on page 95)
  - `lst.snap` (see section 5.8 on page 75)
  - `str.end` (see section 5.17 on page 97)
  - `lst.unique` (see section 5.8 on page 77)
  - `of.type` (see section 5.18 on page 106)

Bug Fixes

- issue with sharing of properties across multiple instances of the same *elemental*
- issue with trigger statement getting missed when frequently repeated
- issue with *primitive* `frm.labels` not unifying correctly with a *list* as second *term*
- issue cloned *elemental* not subscribing to the right label
- crash when re-loading a module
- predicate's limit not considered when collecting variables
- inquiry predicate prefix was not always working
- offloading mode for `MRKCSBFStore` was not working
- div(42,-5) was returning `NaN`

- variable's constraint was lost when unpinning (e.g. with primitive cpy)
- non-working negation of some primitives based predicate
- inconsistent hash code generation for Frame term (when the order of the slot is different)
- `fzz.collect` reusing query's context was causing issue within the *elemental* originator (early failure)
- issue with primitive `frm.labels` and `frm.pairs` not accepting its 2nd term to be a list
- issue with primitive `frm.pairs` not accepting a non-final *list* as second term
- issue with `MRKCCSVStore` always missing the last line

0.7.0-X

Breaking Changes

- predicates:
  - range check or unification to a *variable* of a *predicate*'s *truth value* requieres an `=` character
- terms:
  - *regexp* is no longer an *atom*
  - *escaper* behavior have changed

Changes

- elementals:
  - FZZCFFBNetwork:
    * new `datafile` property to save network to a binary file
    * support for *list* and *data* terms
  - MRKCBFSolver:
    * new property `reply.on` and `cascade`, `cascade.tmo`
  - FZZCCollector
    * speed-up
    * modified behavior of property `tmo` to be the time-out from the last received replies
    * added property `ttl` to specify the time-to-live value for the query
  - MRKCCSVStore
    * property `arity` to specify the arity of the statements (if the number of columns is greater than the arity, the extra will be grouped into a list as the last term)
  - EV3CSENLEGOGyros
    * property `inverted`
  - EV3CBEVSonar
    * support to *peek* at the current reading
- primitives:
  - `lst.sort` now accept as 3rd term a list of indexes to be used for sorting lists (+1 index will be used when the lists' terms are equal)
  - revisited the way the `sim` primitive compute the similarity between two numbers
- samples:
  - updated `irl2asm.fizz`
- terms:
  - a *frame*'s label can be any atom (and not just a *symbol*)
- console:
  - `/spy` output contains the timestamp
  - use `verbose` property to silence output
  - ignore *variables* with name starting with an upper case

Additions

- samples:
  - iris2, iris3
  - nlu
  - movies
  - ml
  - db
  - fuzzy
  - fun, eval, exec, sexp, lstrnd
  - funx, funx2, funx3
  - tasc (based on Hector Levesque's book "Thinking as Computation" (ISBN: 978-0-262-01699-5))
- console:
  - /trace (see section 4.4 on page 38)
- elementals:
  - MRKCStopper (see section 6 on page 115)
  - FZZCFUNRunner (see section 6 on page 116)
  - EV3CSRVMapping (see section 7 on page 138)
- predicates:
  - `?` prefix for *predicate* (see section 2.3 on page 4)
- terms:
  - *data* (see section 3.3 on page 12)
  - *quirk* (see section 3.11 on page 19)
- volatiles ((see section 3.10 on page 18):
  - `sym.8`
  - `sym.6`
  - `now.ms`
- constraints:
  - `fun.label`
- constants:
  - `pi`
- prototypes
  - support for alternate fuzzy and-or evaluation (see section 2.4 on page 7)
- primitives:
  - `rnd.sint` (see section 5.13 on page 89)
  - `qrk.head` (see section 5.12 on page 86)
  - `qrk.tail` (see section 5.12 on page 87)
  - `qrk.make` (see section 5.12 on page 87)
  - `is.quirk` (see section 5.18 on page 105)
  - `lst.any` (see section 5.8 on page 68)
  - `lst.all` (see section 5.8 on page 67)
  - `qat.euler` (see section 5.19 on page 107)

- hush.if.not (see section 5.2 on page 51)
- cut.if (see section 5.2 on page 48)
- cut.if.not (see section 5.2 on page 48)
- lst.min (see section 5.8 on page 73)
- lst.max (see section 5.8 on page 73)
- lst.avg (see section 5.8 on page 68)
- vec.add (see section 5.19 on page 108)
- vec.sub (see section 5.19 on page 110)
- vec.mul (see section 5.19 on page 110)
- vec.div (see section 5.19 on page 109)
- rng.norm (see section 5.14 on page 90)
- daa.find (see section 5.5 on page 60)
- daa.min (see section 5.5 on page 62)
- daa.max (see section 5.5 on page 61)
- daa.avg (see section 5.5 on page 60)
- qat.add (see section 5.19 on page 107)
- qat.sub (see section 5.19 on page 108)
- qat.length (see section 5.19 on page 108)
- lst.it (see section 5.8 on page 71)
- any (see section 5.2 on page 43)

Bug Fixes

- issue with *variables* in the `define` *primitive*
- issue with `fun.terms` not unifying to a `split-list` (as 2nd term)
- issue with property `clone` not finding the *elemental* to clone from
- issue with `rnd.sint` and `rnd.uint` crashing with `"Floating point exception (core dumped)"` when the range was given using the same value
- issue with `then` *primitive* confusing minutes and seconds
- issue with `mao.abs` *primitive* returning `0` when the first term was a negative floating point value
- issue with `frm.make` *primitive* failing when an empty *list* was used as one of the *term*
- issue in `FZZCCLUGateway` leading to long delay in further transmission after a large one

0.6.0-X

Breaking Changes

- `MRKCSBFStore` *elemental class* is impacted by a bug in storing `GUID` *term*.
- Many of the non-core *elementals* have been moved to individual modules (see 7 on page 123).

Changes

- primitives:
  - `str.tokenize` support optional fourth *term* which is a *list* of flags.
  - `peek` accepts a third *term* which is a value to be unified to the 2nd *term* if the label doesn't exists in the properties.
- config:
  - `spinning` meaning changed (see 4.2 on page 23)
- elementals:
  - new `ttl` property to set the time-to-live of any query sent by the *elemental* (instead of using the system default)
  - `MRKCEvaluator`: when no "tmo" is specified, the *substrate* or *elemental* TTL value is used
- predicates:
  - `~` can be used with any label other than `self` (see section 2.3 on page 4).

Additions

- samples:
  - bigrams
  - clu
  - ecalculus
  - robin
- modules:
  - CLU (see section 7 on page 128)
  - EV3 (see section 7 on page 130)
- elementals:
  - constants `$self` and `$guid`
- primitives:
  - `rnd.sint` (see section 5.13 on page 89)
- predicates:
  - `*` prefix for *predicate* (see section 2.3 on page 4)

Bug Fixes

- `guid` *term* wasn't flattened and thus wouldn't get saved in *SBFStore*.
- trigger based *prototypes* where not respecting the 'cut' directives.
- Unfrequent crashes when pasting into the console (outside of the input mode)

0.5.0-X

Breaking Changes

- Pre 0.5 *kindled runtime* (`.bizz`) files can't be loaded
- `MRKCSBFStore` *elemental class* is impacted by hashing changes to *numbers*

Changes

- support for modules (shared library) that can be loaded at runtime (SDK to come in a future release)
- console:
  - previous query is no longer cancelled when a new one is issued
  - query specified via the *command line* gets executed once all the files specified in the *command line* have been loaded
- new *elemental* properties:
  - `chatty` (see section 2.5 on page 8)
  - `noisy` (see section 2.5 on page 8)
  - `clone` (see section 2.5 on page 8)
- any *elemental* property can be read using the *constant* syntax
- new property for *elemental* of class `MRKCBSSolver`:
  - `memoize` (see. fibonacci sample)
- new property for *elemental* of class `MRKCLettered`:
  - `recall.frq`, `recall.ttl`, `recall.add`, `recall.mul`, `recall.thd` (see section 6 on page 122)
- *primitives* `gt`, `gte`, `lt` and `lte` now also works with *strings* and *symbols*

Additions

- *solution* files (see section 4.3 on page 26)
- new console *command*: `/use` (see section 4.4 on page 39)
- new *syntax*:
  - `~` prefix for *predicate* (see section 2.3 on page 4)
  - `self` *predicate* (see section 2.3 on page 4)
- new *terms*:
  - *regexp* (see section ?? on page ??)
- new *primitives*:
  - `frm.erase` (see section 5.6 on page 62)
  - `lst.mix` (see section 5.8 on page 74)
  - `lst.sort` (see section 5.8 on page 76)
  - `lst.sub` (see section 5.8 on page 77)
  - `rex.make` (see section 5.15 on page 93)
  - `rex.match` (see section 5.15 on page 93)
  - `rng.rand` (see section 5.14 on page 92)
- new *constraints*:
  - `eq`
  - `is.regexp`
  - `is.bound`
- new *volatiles*: `sym.3`, `sym.4` and `sym.10` (see section 3.10 on page 18)

Bug Fixes

- `lst.item`, `lst.head`, `lst.tail` would not unify theirs last *term* with a *list*.
- `MRKCTicker` wouldn't accept a property as a *constant*.
- `peek(guid,:x)` was unifying `:x` with a *string* instead of a *guid*.
- `frm.fetch(a = [1,2],a,[_,:v])` wasn't returning `2`.
- re-saving an *elemental* into a *fizz* file was failing.
- *terms* in a *range* couldn't be a *constant*.
- the hashcode of real *number* was the same regardless of the sign.
- `lst.tail` was not unifying its second *term* with `[]` when the first *term* was an empty *list*.

0.4.0-X

Additions

- new *elementals*:
  - `MRKCSBFStore` (see section 6 on page 115)
  - `MRKCCSVStore` (see section 6 on page 113)
  - `FZZCLGRProcessor` (see section 7 on page 123)
- new *terms*:
  - *guid* (see section 3.1.5 on page 11)
- new *primitives*:
  - `str.trim.head` (see section 5.17 on page 101)
  - `str.trim.tail` (see section 5.17 on page 101)
  - `str.tail` (see section 5.17 on page 99)
  - `str.head` (see section 5.17 on page 98)
  - `lst.incl` (see section 5.8 on page 71)
  - `lst.excl` (see section 5.8 on page 70)
  - `lst.join` (see section 5.8 on page 72)
  - `lst.init` (see section 5.8 on page 71)
  - `sym.cmp` (see section 5.16 on page 94)
  - `sim` (see section 5.1 on page 42)
  - `is.even` (see section 5.18 on page 103)
  - `is.odd` (see section 5.18 on page 104)
  - `gid.make` (see section 5.11 on page 84)
- new *constraints*:
  - `lst.incl`
  - `lst.excl`
  - `is.guid`
  - `is.even`
  - `is.odd`

Changes

- modified *primitives*:
  - `lst.remove` was changed to succeed when the item to remove isn't found in the *list*.
  - `str.trim` was changed to accept an optional third *term*: the *string* to be trimmed from the 1st *term*.
  - `lst.length` was changed to accept a third *term* which is the *term* to be assigned to each of the *list's terms* when the first *term* of the primitive is an *unbound variable*.
  - `fzz.lst` was changed to returns a *list* of *guid terms* instead of a *list* of *strings*.
  - `guid.str` and `guid.sym` were renamed `gid.str` and `gid.sym`.
- modified *console commands*:
  - `/peek` now accepts a *guid*.
  - `/poke` now accepts a *guid*.
  - `/tells` now accepts a *guid* as well as a *symbol*.
  - `/knows` now accepts a *guid*.
- modified *terms*:
  - *binary* syntax has changed to single quote *functor*.
  - *symbol* can now include `+` or `*` as long as they are not on the first character.

Bug Fixes

- *constraint* `is.string` was testing for a variable to be bound to a *symbol*
- *primitive* `str.swap` in some condition was repeating part of the tail of the *string* where the replacement was occuring
- *primitive* `add` was returning 0 when used with an unsigned number as the first term and a negative number as the second term (e.g. `add(23u,-18,:v)`)
- *string terms* with control characters were not rendered properly when they are embedded in other terms

0.3.0-X

Additions

- *live code reload* functionality
- new *constant* `$cores`
- new *primitives*:
  - `aeq` (see section 5.3 on page 57)
  - `bundle` (see section 5.2 on page 44)
  - `div.int` (see section 5.1 on page 40)
  - `fzz.lst` (see section 5.11 on page 84)
  - `lst.remove` (see section 5.8 on page 74)
  - `mao.sign` (see section 5.10 on page 82)
  - `str.find` (see section 5.17 on page 97)
  - `str.flip` (see section 5.17 on page 97)
  - `str.trim` (see section 5.17 on page 101)
  - `str.rest` (see section 5.17 on page 98)
  - `str.swap` (see section 5.17 on page 99)
  - `sym.cat` (see section 5.16 on page 93)
- new *console commands*:
  - `/reload` (see section 4.4 on page 36)
  - `/import.txt` (see section 4.4 on page 33)
- new *class* `FZZCWebAPIGetter` (see section 7 on page 125)

Changes

- increased the maximum number of threads that can be used by the console
- added support for `str.find` as a *variable's constraint*
- *primitive* `frm.fetch` allows for a fourth *term* to specify a default value to use if the label isn't found
- when the first *term* of the `/peek` and `/poke` *console commands* is a *symbol*, all *elemental* of that label will be targetted
- the `fzz.eval` service now accept a *list* as second *term* to describe the *functor* to be evaluated
- changed *class* `FZZCTicker` to support the property `tick.on.attach`
- changed *class* `MRKCBFSolver` to support the property `replies.are.triggers`
- changed *class* `MRKCLettered` to support the property `nearest.only`

Bug Fixes

- minor performance tweaks when parsing *list* in *fizz* source files
- *primitive* `str.sub` was not properly handling negative offset
- on occasion queries/replies where not being sent/received
- JSON support wasn't handling 'null' value (causing crash)
- chunked transfer encoding wasn't supported by the builtin web client

0.2.0-X

Additions

- added console commands `/import.json` and `/export.json` to import and export JSON files (see section 4.4 on page 32 and 4.4 on page 29)
- added *primitive* `change` (see section 5.2 on page 47)
- added *primitive* `console.exec` (see section 5.2 on page 47)
- added *primitive* `then` (see section 5.2 on page 56)
- added *primitive* `tme.str` (see section 5.2 on page 56)
- added *primitive* `str.cmp` (see section 5.17 on page 96)
- added *elemental* class `FZZCWebAPIPuller` for fetching JSON data from web services (see section 7 on page 127)

Changes

- console commands `/import` and `/export` were renamed `/import.csv` and `/export.csv`
- the *elemental* class `FZZCTicker` now also supports time interval expressed in seconds (see section 6 on page 121)

Bug Fixes

- published statements could stop from being received by *elementals* referencing them as trigger
- *primitive* `str.tosym` was failing when the first *term* was already a *symbol*

0.1.4-X

Changes

Initial Release

Bug Fixes

Initial Release

Known issues

- Poor performance with *inferences* that involves *combinatorial exploration*
- Parser's error handling is too terse
- An empty comment line will cause a parsing error in a `fizz` file

Index