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.

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 2

2 Concepts & Syntax 2
  2.1 Knowledge ................................................. 2
  2.2 Statement ................................................. 3
  2.3 Predicate ............................................... 4
  2.4 Prototype ............................................... 6
  2.5 Elemental ............................................... 7
  2.6 Service ................................................ 8

3 Terms 8
  3.1 Atoms .................................................. 8
  3.2 List .................................................... 12
  3.3 Frame ................................................... 12
  3.4 Functor .................................................. 13
  3.5 Range .................................................... 13
  3.6 Variable ................................................ 13
  3.7 Constant ............................................... 15
  3.8 Volatile ............................................... 15

4 Console 16
  4.1 Usage ................................................... 16
  4.2 Adjusting the runtime ................................. 17
  4.3 Solution ................................................ 19
  4.4 Commands .............................................. 20

5 Primitives 33
  5.1 Arithmetic ............................................. 33
  5.2 Basic .................................................. 36
  5.3 Comparisons .......................................... 44
  5.4 Frame ................................................... 46
  5.5 Functor .................................................. 49
  5.6 List ..................................................... 50
  5.7 Boolean Logic ......................................... 56
  5.8 Mathematics .......................................... 58
  5.9 Miscellaneous ........................................ 61
  5.10 Random ................................................ 62
  5.11 Range .................................................. 63
  5.12 Regexp ................................................ 66
  5.13 Symbol ................................................ 67
  5.14 String .................................................. 68
  5.15 Typing ................................................ 73

6 Elementals 76

7 Advanced topics 87

Index 96
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 built-in 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:

---

1 not yet fully implemented
**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 3.3 on page 12 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 parentheses and terminated by a semicolon. We will look into all the supported terms in more details in Section 3 on page 8, 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 occurrence 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 is assigned a value to represent the likelihood of a given weather occurrence in a particular city having been timestamped (see section 5.2 on page 36 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;
}
```
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 33 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 environment 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>

There is a third prefix that can be used ” for a predicate in conjunction with the predicate label self. This indicates a self referencing predicate (a recursive predicate). While using the full name of the elemental will
also work, 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.

Lastly, if a caret (^) is added right after the *terms* of the *predicate*, it will indicate that once the *predicate* as succeeded, 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>
3> -> ( "b" ) := 1.00 (0.001) 1
```

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>
4> -> ( "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 3.6 on page 13 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:
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, let's backtrack to the following example:

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

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

## 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 76 for more details. Each elemental presents on the substrate is assigned a 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 3.3 on page 12) in between the knowledge's body and its label, as seen in the following example:

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:
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 41) or by using the constant access syntax (e.g. $guid).

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 unnecessary 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 7 on page 88 for more details.

3 Terms

There are eight 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
- Regexp

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:

```plaintext
eyearly_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 17).

3.1.2 String

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

```plaintext
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*:

```plaintext
identifiers {
}
```
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:

```fizz
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:

```fizz
guids {
  ('guid("71cfade6-3cab-c34e-3ca6-e7a43e6fb5f7");
}
```

### 3.1.6 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:

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

As *fizz* uses the PCRE2 library\(^2\) 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:

\(^2\)see https://www.pcre.org/
The **CASELESS** flag can be used to ignore the case during matching (note that flags are case-insensitive):

```prolog
?- 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:

```prolog
1 str.is {  
2 3  ('regexp("[+-]?([0-9]*[.])?[0-9]+"),number)~ :- true;  
4  (_,:string) :- true;
5 6 }
```

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

```prolog
1 test {  
2 3  (:s?[rex.match('regexp("(the|a)?\s?(dog|cat)\sis\swet|cold|sick)"),:s,[_,_,,:c]],:c) :- true;
4 5 }
```

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

```fizz
1 color {
2  (red,[1.0,0.0,0.0]);
3  (green,[0.0,1.0,0.0]);
4  (blue,[0.0,0.0,1.0]);
5 }
```

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:

```fizz
1 lst.print {
2  ([]);
3  ([:h]:r) :- console.puts(:h), @lst.print(:r);
4 }
```

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 contains 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 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:

```fizz
1 gameboy.color {
2  ({r = 0.509803, g = 0.784313, b = 0.294117});
3  ({r = 0.325490, g = 0.670588, b = 0.392156});
4  ({r = 0.164705, g = 0.549019, b = 0.349019});
5  ({r = 0.000000, g = 0.294117, b = 0.282352});
6 }
```

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 symbol. Unlike with lists, unification of two frames will only be done over the labels that both terms have in common.
3.4 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:

```plaintext
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.

3.5 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:

```plaintext
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)`.

3.6 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 across 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
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,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

*Constraints* are specified after a variable with a question mark followed by list or a variable which will be bound at runtime to a list. Each of the element in the list (which can be a functor, range or symbol) 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:

<table>
<thead>
<tr>
<th>Functor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gt</td>
<td>greater than</td>
</tr>
<tr>
<td>gte</td>
<td>greater than or equal</td>
</tr>
<tr>
<td>lt</td>
<td>lesser than</td>
</tr>
<tr>
<td>lte</td>
<td>lesser than or equal</td>
</tr>
<tr>
<td>neq</td>
<td>not equal</td>
</tr>
<tr>
<td>aeq</td>
<td>almost equal</td>
</tr>
<tr>
<td>eq</td>
<td>equal/unify</td>
</tr>
<tr>
<td>lst.member</td>
<td>value is present in a list</td>
</tr>
<tr>
<td>lst.except</td>
<td>value is not present in a list</td>
</tr>
<tr>
<td>lst.incl</td>
<td>value is a list that include the items in a list</td>
</tr>
<tr>
<td>lst.excl</td>
<td>value is a list that exclude the items in a list</td>
</tr>
<tr>
<td>is.atom</td>
<td>value is an atom term</td>
</tr>
<tr>
<td>is.binary</td>
<td>value is a binary term</td>
</tr>
<tr>
<td>is.string</td>
<td>value is a string term</td>
</tr>
<tr>
<td>is.symbol</td>
<td>value is a symbol term</td>
</tr>
<tr>
<td>is.number</td>
<td>value is a number term</td>
</tr>
<tr>
<td>is.regexp</td>
<td>value is a regexp term</td>
</tr>
<tr>
<td>is.guid</td>
<td>value is a guid term</td>
</tr>
<tr>
<td>is.list</td>
<td>value is a list term</td>
</tr>
<tr>
<td>is.range</td>
<td>value is a range term</td>
</tr>
<tr>
<td>is.frame</td>
<td>value is a frame term</td>
</tr>
<tr>
<td>is.func</td>
<td>value is a functor term</td>
</tr>
<tr>
<td>is.bound</td>
<td>a value is bound</td>
</tr>
<tr>
<td>is.unbound</td>
<td>no value is bound yet</td>
</tr>
<tr>
<td>is.even</td>
<td>value is an even number</td>
</tr>
<tr>
<td>is.odd</td>
<td>value is an odd number</td>
</tr>
<tr>
<td>str.find</td>
<td>value is a string which contains a specified substring</td>
</tr>
</tbody>
</table>

14
Most functors require a single term except the is.* ones which can be given as a symbol, and aeq which expects two. Constraints can be used on any variables, including in a prototype’s entrypoint as shown here:

```prolog
lst.zip {
  ([],[])^ :- true;
  ([[:e],[:e]])^ :- true;
  ([[:e,:e:r]:1) :- #lst.zip([[:e]:r],[:e]:1);
  ([[:e,:f?neq(:e)]:r],[[:e]:l]) :- #lst.zip([[:f]:r],[:e]: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.

### 3.7 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`

### 3.8 Volatile

Volatile 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:

```prolog
?- 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
- `%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.10` a randomly generated symbol of 10 characters length
- `%gui` a randomly generated GUID as a string

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

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:

```javascript
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

Several 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 extension `.json`. Here’s an example of a file that adjusts all the possible parameters:

```json
{
  "runtime" : {
    "scheduler" : {
      "threads" : 4,
      "affinity" : true,
      "spinning" : 500
    },
    "offloader" : {
      "minpool" : 1,
      "maxpool" : 4,
      "timeout" : 750,
      "affinity" : false
    },
    "httpclient" : {
      "dnstimeout" : 3.5,
      "maxconnect" : 8,
      "maxrequest" : 4,
      "maxresolve" : 4,
      "maxcontent" : 2048
    },
    "livereload" : {
      "enabled" : true,
      "interval" : 250
    }
  },
  "substrate" : {
    "ttl" : {
      "type" : "real",
      "data" : 55.0
    },
    "grace" : {
      "type" : "real",
      "data" : 0.5
    },
    "sspr" : {
      "type" : "uint",
      "data" : 8
    }
  }
}
```
It contains two sections: the runtime and the substrate. The former adjusts the threading and multi-cores models of the runtime while the later adjusts the common behavior of all elemental objects will use.

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

threads represents the number of threads to be used. 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 amount of time (in us) that each thread will spin for when waiting to something to do before going to sleep. Increasing that value may lower the latency at the cost of a higher CPU load.

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 httpclient section is responsible for tuning the built-in HTTP client used by the elemental class FZZCHttpPuller. If this section is not present in the configuration file, the HTTP client will not be available. The key/value pairs meanings is as follows:

dnstimeout timeout (in seconds) when performing a DNS lookup.
maxconnect maximum number of concurrent connection to any host.
maxrequest maximum number of concurrent request for the same host (0 for no limit).
maxresolve maximum number of concurrent Domain-name resolution (0 for default).
maxcontent maximum size of the content to store in RAM, before storing it into a temporary file.
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 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 milliseconds) at which each elemental 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).

Lastly, there are two command line options of interest: `-s` and `-c`. The foremost 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:
To be valid, such file must contain 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)
- **globals** a list of objects describing the constants to be created. Each of the objects must contain 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:

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

Close the console and terminate the executable.
create

 DEFINE 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

 DEFINE 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

 DEFINE 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

 DEFINE 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:
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
```
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*:

```ILOG
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 is in between 0.1 and 0.4, we would do:

```ILOG
?- /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:

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

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:

```JSON
{
  "r" : 0.325490,
}
```
When the statements to be exported do not contain a single term, all the exportable terms will be exported within a JSON array. For example, if we consider the following knowledge:

```prolog
product {
  (model_e, tesla, 2012);
  (iphone_x, apple, 2018);
  (vive, htc, 2015);
  (coconut_water, zico, 2000);
}
```

and export it as follow:

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

The JSON file will then contain:

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

`freeze` is a command that freezes the runtime environment 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` clears the console's history.
Change the length of the console’s history. The default is 100.

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 the fourth term contains the number of each column (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:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.1</td>
<td>3.5</td>
<td>1.4</td>
<td>0.2</td>
<td>Iris-setosa</td>
</tr>
<tr>
<td>2</td>
<td>4.9</td>
<td>3.0</td>
<td>1.4</td>
<td>0.2</td>
<td>Iris-setosa</td>
</tr>
<tr>
<td>3</td>
<td>7.0</td>
<td>3.2</td>
<td>4.7</td>
<td>1.4</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>4</td>
<td>6.4</td>
<td>3.2</td>
<td>4.5</td>
<td>1.5</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>5</td>
<td>6.3</td>
<td>3.3</td>
<td>6.0</td>
<td>2.5</td>
<td>Iris-virginica</td>
</tr>
<tr>
<td>6</td>
<td>5.8</td>
<td>2.7</td>
<td>5.1</td>
<td>1.9</td>
<td>Iris-virginica</td>
</tr>
</tbody>
</table>

We would do as follows:

```prolog
?- /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(7.0, 3.200000, 4.700000, 1.400000, "Iris-versicolor") := 1.00
spy : S iris(6.4, 3.200000, 4.500000, 1.500000, "Iris-versicolor") := 1.00
spy : S iris(6.3, 3.3, 6.0, 2.5, "Iris-virginica") := 1.00
spy : S iris(5.8, 2.7, 5.1, 1.9, "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:

```prolog
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. For the sake of simplicity, the JSON file below was abbreviated:

```json
{
  "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:

?- /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)

---

3 http://api.fixer.io/latest?base=USD
The code in `fixer.fizz` splits the work over two `elementals`: `process` and `process.rates`:

```fizz
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`, fetches from the `frame` it contains the value for the `base` and `rates` labels and pass them to the second `elemental`:

```fizz
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 fetches 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:

```fizz
?- /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, "ACCATTACCACAGGTACGGTGCGGGCTGA...GAAAAAAGCCCGCACCTGACAGTGCGGG") := 1.00 (700.000000)
```
import.txt: 10 lines read in 0.001s.

**kindle**

/\kindle\(\text{string}\)\)

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\(\text{symbol}|\text{string}|\text{guid}\)\)

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
}
```

We can then use that alias with the \knows\ command:

```
?- /\knows\(\text{c.range}\)
no
?- /\knows\(\text{crange}\)
yes
```
This command generates a list of all the *elemental* objects present on the *substrate*. Each of the output lines will contain, in order, the GUID, the class, label and, if available, the alias of each *elementals*:

```
?- /list

list : 288a77db-bab2-1748-38af-892fclf18d112 MRKCLettered blobs
list : bf006e31-4bd4-c348-a1a7-0449fb0a167f MRKCLettered car.range (crange)
list : 1bb328bb-4938-8a43-9db0-2a1e655acc19b MRKCLettered color
list : 3cfc2da3-8728-0d49-22a0-761d19af28bb MRKCLettered gameboy.color
list : c9928201-4dbd-5e4d-bab7-ee9e13c771dc MRKCLettered identifiers
list : 47d3366d-5794-0949-d7a4-f7e462dafaa4 MRKCBFSolver lst.print
list : 966b0df2-7010-6542-5f83-5cedb64afadb MRKCBFSolver maybe_rainbow
list : 4048e6be-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-c537d7fbb117c 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 : cb6a0d33-0000-0644-8f9a-c7e678060aff MRKCLettered weather2
list : 45f63bd5-b824-594f-e990-148724ef64d MRKCLettered yearly_stats
list : 17 elementals listed in 0.000s
```

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 32) to manually unload the *knowledge* from a given set of files.

The `reload` command allows *knowledge* to be re-loaded from (properly formatted) text files. All terms in the *predicate* are expected to be *strings*.
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 7, we can change the value of its `min` properties as follows:

```prolog
?- /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.

The `save` command allows knowledge to be saved to a (properly formatted) text file, allowing it to be reloaded 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:

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

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

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

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

The `scan` command will keep printing statistics on the runtime environment until none of the statistics changes in the substrate:
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 extremely 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(0.100000), lt(0.400000)], g = :g, b = :b})
spy : Q @gameboy.color({r = :r ? [gt(0.100000), lt(0.400000)], g = :g, b = :b}) (14.999830)
-> ( 0.325490 , 0.670588 , 0.392156 ) := 1.00 (0.001) 1
spy : R gameboy.color({r = 0.164705, g = 0.549019, b = 0.349019}) := 1.00 (14.999510)
-> ( 0.164705 , 0.549019 , 0.349019 ) := 1.00 (0.001) 2
```

Output from *spying* will always be prefixed with `spy`. The first letter after the colon 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 *knowledge* 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 milliseconds) 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**

/\texttt{tells(symbol\mid string\mid guid,functor\mid 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.

?- /\texttt{tells(some.obj,do(this,45))}

**unload**

/\texttt{unload(string+)}

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

?- /\texttt{load("./samples/manual.fizz")}

load : loading ./samples/manual.fizz ...  
load : loaded ./samples/manual.fizz in 0.011s  
?- @\texttt{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  
?- /\texttt{unload("./samples/manual.fizz")}

unload : unloading ./samples/manual.fizz ...  
unload : unloaded ./samples/manual.fizz in 0.000s

**use**

/\texttt{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.

?- /\texttt{use("modLGR")}

use : loading ./mod/linux/x64/modLGR.so ...  
use : loaded ./mod/linux/x64/modLGR.so in 0.001s  
?- /\texttt{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.
The `wipe` command will cause the runtime environment 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.

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 7, 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.

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add</code></td>
<td>This <code>primitive</code> will unify or bind the sum of its two first <code>terms</code> with the third. For example:</td>
</tr>
<tr>
<td></td>
<td><code>- add(4,3,:x)</code></td>
</tr>
<tr>
<td></td>
<td><code>-&gt; ( 7 ) := 1.00 (0.001) 1</code></td>
</tr>
</tbody>
</table>

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
```

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>div</code></td>
<td>This <code>primitive</code> will unify or bind the division of the first <code>term</code> by the second with the third. For example:</td>
</tr>
<tr>
<td></td>
<td><code>- div(10,3,:x)</code></td>
</tr>
<tr>
<td></td>
<td><code>-&gt; ( 3.333333 ) := 1.00 (0.000) 1</code></td>
</tr>
</tbody>
</table>
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

**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)
\[\rightarrow (30) := 1.00 (0.000) 1\]

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)
\[\rightarrow (0.400000) := 1.00 (0.000) 1\]

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)
\[\rightarrow (0.785714) := 1.00 (0.000) 1\]

?- sim(3.21,10,:s)
\[\rightarrow (-0.743261) := 1.00 (0.000) 1\]

?- sim(3.21,-100,:s)
\[\rightarrow (-0.980808) := 1.00 (0.000) 1\]

?- sim(3.21,2.211,:s)
\[\rightarrow (0.000500) := 1.00 (0.000) 1\]

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)
\[\rightarrow (6) := 1.00 (0.000) 1\]

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)
\[\rightarrow (6) := 1.00 (0.000) 1\]
sum

`sum(number+, number|variable)`

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

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

Contrary 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*.

**assert**

```prolog
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*:

```prolog
?- @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:

```prolog
?- 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.

```prolog
?- 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.

```prolog
?- 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:

```prolog
1 import.frag {
2   ( ) :- @line.f(:i,:s), bundle(frag(:i,:s),1,{},1024), hush;
3 }
```

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

```prolog
?- 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`

```prolog
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:

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

### `console.gets`

```prolog
console.gets(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. For example:

```prolog
?- &console.gets(:x)
>= hello world!
-> ( "hello world!" ) := 1.00 (5.105) 1
```

### `console.puts`

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

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

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

### `declare`

```prolog
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 required 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:

```java
list.print {
  ([[]) ^ := true;
  ([:h| :t]) := console.puts(:h), #lst.print(:t);
}
```

39
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 definitions 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 42.

```
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
```

```
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.

```
own
nownumber|variable)
```

This primitive will unify and/or substitute its sole term with the current host time (UTC, expressed in seconds since Unix epoch).
The `peek primitive` allows for a `property` of the calling `elemental` object to be read and unified and/or substituted 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`:

```prolog
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.

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:

```prolog
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),
```

41
Just like with the `peek` primitive, offloading the execution of the `primitive` will not work.

```
`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.

```prolog
?- @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 39). 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`.

```
`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
```
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

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

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 `milliseconds`. 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

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:
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.

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

\textit{lte}(\textit{term},\textit{term})

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

neq

\textit{neq}(\textit{term},\textit{term})

\textit{neq}(\textit{term},\textit{term}, \textit{boolean}\mid \textit{variable})

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

?- neq(3,5,:e)
\texttt{-> ( true ) := 1.00 (0.000) 1}
?- neq(3,3,:e)
\texttt{-> ( false ) := 1.00 (0.000) 1}

5.4 Frame

All \textit{primitives} related to handling \textit{frames} are grouped in this category.

frm.erase

\textit{frm.erase}(\textit{frame},\textit{symbol}\mid \textit{variable})

\textit{frm.erase}(\textit{frame},\textit{symbol}\mid \textit{variable},\textit{term}\mid \textit{variable})

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

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

frm.fetch

\textit{frm.fetch}(\textit{frame},\textit{symbol}\mid \textit{variable},\textit{term}\mid \textit{variable})

\textit{frm.fetch}(\textit{frame},\textit{symbol}\mid \textit{variable},\textit{term}\mid \textit{variable},\textit{term})

The \textit{primitive} \textit{frm.fetch} main purpose is to get the value stored in a frame (the first \textit{term}) for a given label (the second \textit{term}) and unify it with the third \textit{term}. If a fourth \textit{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 \textit{frame}. For example:

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

If the second \textit{term} is an unbound variable, the inference engine will list all label/value combinations:

?- frm.fetch({a = 3, b = hello},1,:v)
\texttt{-> ( a , 3 ) := 1.00 (0.000) 1}
\texttt{-> ( b , hello ) := 1.00 (0.001) 2}
**frm.length**

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

This *primitive* will unify or substitute 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 substitute 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 substitutes 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

47
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 substitute 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 substitute 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
The primitive `frm.pairs` will unify or substitute 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.pairs({a=1,b=2,c=3},[a,c],:pairs)
-> ([a, 1], [c, 3]) := 1.00 (0.000) 1
```

This primitive will merge two frames and unify/replace 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
```

This primitive will extract a collection of label/value pairs from the frame given as the first term and unify or substitute 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
```

### 5.5 Functor

This section covers all the primitives that manipulate functors.

The primitive `fun.length` will unify or substitute 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
```
The `fun.make` primitive unify or substitute 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

This `fun.member` 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

This `fun.label` primitive will unify or substitute its second term with the label of the `functor` passed as first term.

This `fun.terms` primitive will unify or substitute 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.

### 5.6 List

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

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

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

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

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.flip**

\( \text{lst.flip(list\ variable, list\ variable)} \)

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**

\( \text{lst.head(list, term)} \)

This primitive will unify or substitute 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**

\( \text{lst.incl(list, list)} \)

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**

\( \text{lst.init(list, list\ variable)} \)

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.item**

\[
\text{lst.item}(\text{list, number|variable, term|variable})
\]

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

\[
\text{lst.join}(\text{list, list, list| variable})
\]

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.length**

\[
\text{lst.length}(\text{list, number|variable})
\]

This primitive will unify or substitute 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 wildcard 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 wildcard variable. For example:

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

\[
\text{lst.make}(\text{term}, \text{list|variable})
\]

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.member**

\[
\text{lst.member}(\text{term|variable}, \text{list|variable})
\]

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.mix**

\[
\text{lst.mix}\left(\text{list}, \text{list|variable}\right)
\]

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.remove**

\[
\text{lst.remove}(\text{term}, \text{list}, \text{list|variable})
\]

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 substitute 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

54
**lst.rest**

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

This *primitive* will unify or substitute 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.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],:1)
\(\rightarrow ( [1, 3, 3, 4, 7, 9] ) := 1.00 (0.001) 1\)
?- lst.sort([[3,a],[7,b],[1,d],[9,f],[4,e],[3,z]],:1,1)
\(\rightarrow ( [3, a], [7, b], [1, d], [4, e], [9, f], [3, z] ) := 1.00 (0.001) 1\)
```

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

**lst.sub**

```
 lst.sub(list|variable, number, number, list|variable)
```

The **lst.sub** **primitive** will unify or substitute 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])
\(\rightarrow ( 6 ) := 1.00 (0.000) 1\)
```

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 occurrences 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]])
\(\rightarrow ( 4, 2 ) := 1.00 (0.001) 1\)
\(\rightarrow ( 7, 2 ) := 1.00 (0.001) 2\)
```

**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)
\(\rightarrow ( [f, b, c, d, e] ) := 1.00 (0.001) 1\)
?- lst.swap([a,b,c,d,e],3,f,:l)
\(\rightarrow ( [a, b, c, f, e] ) := 1.00 (0.001) 1\)
```

**lst.tail**

```
 lst.tail(list, list|variable)
```

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

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

### 5.7 Boolean Logic

This section contains all the **primitives** that deal with boolean logic operations.
boo.and

\texttt{boo.and(boolean+, boolean| variable)}

This \textit{primitive} will unify or bind its last \textit{term} with the boolean \texttt{AND} of all other \textit{terms}. For example:

\begin{verbatim}
?- boo.and(1,0,1,:v)
\rightarrow (\ false \ ) := 1.00 \ (0.000) \ 1
?- boo.and(1,1,1,:v)
\rightarrow (\ true \ ) := 1.00 \ (0.000) \ 1
\end{verbatim}

boo.not

\texttt{boo.not(boolean| variable, boolean| variable)}

This \textit{primitive} will unify or bind its \textit{terms} with the boolean negation of the other \textit{term}. For example:

\begin{verbatim}
?- boo.not(1,:v)
\rightarrow (\ false \ ) := 1.00 \ (0.001) \ 1
?- boo.not(0,:v)
\rightarrow (\ true \ ) := 1.00 \ (0.000) \ 1
?- boo.not(0,1)
\rightarrow (\ ) := 1.00 \ (0.001) \ 1
?- boo.not(:v,1)
\rightarrow (\ false \ ) := 1.00 \ (0.000) \ 1
\end{verbatim}

boo.or

\texttt{boo.or(boolean+, boolean| variable)}

This \textit{primitive} will unify or bind its last \textit{term} with the boolean \texttt{OR} of all other \textit{terms}. For example:

\begin{verbatim}
?- boo.or(1,0,1,:v)
\rightarrow (\ true \ ) := 1.00 \ (0.000) \ 1
?- boo.or(1,1,1,:v)
\rightarrow (\ true \ ) := 1.00 \ (0.000) \ 1
?- boo.or(0,0,:v)
\rightarrow (\ false \ ) := 1.00 \ (0.000) \ 1
\end{verbatim}

boo.xor

\texttt{boo.xor(boolean+, boolean| variable)}

This \textit{primitive} will unify or bind its last \textit{term} with the boolean \textit{exclusive disjunction} of all other \textit{terms}. For example:

\begin{verbatim}
?- boo.xor(1,1,:v)
\rightarrow (\ false \ ) := 1.00 \ (0.001) \ 1
?- boo.xor(1,0,:v)
\rightarrow (\ true \ ) := 1.00 \ (0.000) \ 1
?- boo.xor(1,0,1,:v)
\rightarrow (\ false \ ) := 1.00 \ (0.001) \ 1
?- boo.xor(1,0,0,:v)
\rightarrow (\ true \ ) := 1.00 \ (0.001) \ 1
\end{verbatim}
5.8 Mathematics

This section contains all the primitives that deal with mathematical operations.

**mao.abs**

\[ \text{mao.abs}( \text{number} | \text{variable}, \text{number} | \text{variable}) \]

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 generate two statements. For example:

```prolog
?- 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.ceil**

\[ \text{mao.ceil}( \text{number} | \text{variable}, \text{number} | \text{variable}) \]

This primitive will unify or bind the second term with the smallest integer value greater than or equal to the first term. For example:

```prolog
?- 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:

```prolog
?- mao.ceil(:r,3)
-> ( <2.000001|2.999999> ) := 1.00 (0.000) 1
```

**mao.exp**

\[ \text{mao.exp}( \text{number} | \text{variable}, \text{number} | \text{variable}) \]

This primitive will unify or bind the second term with \( e \) raised to the power of the first term. For example:

```prolog
?- 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:

```prolog
?- 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:

```prolog
?- 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 point value created from the integer and fractional values:

```prolog
?- 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:

```prolog
?- 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 (inverse power). For example:

```prolog
?- 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:

```prolog
?- 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**

\[
\text{mao.sign}(\text{number}, \text{number} | \text{variable})
\]

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.sqrt**

\[
\text{mao.sqrt}(\text{number} | \text{variable}, \text{number} | \text{variable})
\]

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 be computed:

?- mao.sqrt(:v,5)
-> ( 25 ) := 1.00 (0.000) 1

### 5.9 Miscellaneous

**fzz.lst**

\[
\text{fzz.lst}(\text{variable} | \text{list})
\]

\[
\text{fzz.lst}(\text{symbol}, \text{variable} | \text{list})
\]

This *primitive* will unify its 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)
-> ( [\text{guid}("263e7d79-c5e4-1f48-6a99-c8c022a2dbf3"), \text{guid}("1e9618ff-8e93-0147-efb6-5527b88c99cb"), \text{guid}("8cb3f7f9-6456-d94b-3393-8766fb3d4c72"), \text{guid}("f4608c21-6a8f-ab4c-4d92-5b092fa4171e"), \text{guid}("40b3f684-f545-0241-99be-998167b99ab6"), \text{guid}("a099afdb-93a7-db4c-40b8-0341ea987ed9"), \text{guid}("330a4f04-e64c-8949-ec9e-83490c365dcb"), \text{guid}("71cfade6-3cab-3c4f-3ca6-e7a43f6b5f7f"), \text{guid}("91448e7-0dd7-bc43-2795-3ec6e3a71537"), \text{guid}("2b5a6f6f-7b4b-394b-dfa5-261d8c07a6f6")]) := 1.00 (0.002) 1
\textbf{gid.make}

\texttt{gid.make(guid|variable)}

This \textit{primitive} will unify or substitute its only \textit{term} with a randomly generated \textit{guid term}. Here’s an example:

?- \texttt{gid.make(:g)}
\rightarrow \texttt{( 'guid("e30f998a-020d-fd4c-c0b8-e384d2dc8020")' ) := 1.00 (0.001) 1}
?- \texttt{gid.make(:g)}
\rightarrow \texttt{( 'guid("ce0c25e6-5adc-9e48-0c80-57b70db9a2e0")' ) := 1.00 (0.000) 1}

\textbf{gid.sym}

\texttt{gid.sym(symbol|variable)}

This \textit{primitive} will unify or substitute its \textit{term} with a randomly generated \textit{symbol}. Here’s an example:

?- \texttt{gid.sym(:g)}
\rightarrow \texttt{( yzrzqbtaxcqrerbuyeaqcfuysbfuw ) := 1.00 (0.000) 1}

The generated symbol is a \textit{globally unique identifier} (GUID).

\textbf{gid.str}

\texttt{gid.str(symbol|variable)}

This \textit{primitive} will unify or substitute its \textit{term} with a randomly generated \textit{string}. Here’s an example:

?- \texttt{gid.str(:g)}
\rightarrow \texttt{( "005a7ce9-433f-574c-d1ba-5a03240eb98e" ) := 1.00 (0.000) 1}

\section*{5.10 Random}

This section describes \textit{primitives} that generate random \textit{numbers}.

\textbf{rnd.real}

\texttt{rnd.real(number,number|variable,number?,number?)}

This \textit{primitive} will unify or bind the second \textit{term} with a series of (floating point) random \textit{number} picked in the range defined in between the third and fourth \textit{terms}. The first \textit{term} is the count of random \textit{numbers} to be provided. For example:

?- \texttt{rnd.real(5,:v,1,100)}
\rightarrow \texttt{( 86.598612 ) := 1.00 (0.000) 1}
\rightarrow \texttt{( 80.759627 ) := 1.00 (0.000) 2}
\rightarrow \texttt{( 41.959139 ) := 1.00 (0.000) 3}
\rightarrow \texttt{( 30.452654 ) := 1.00 (0.001) 4}
\rightarrow \texttt{( 20.528467 ) := 1.00 (0.001) 5}

When no range is provided, the random number will all be in between 0 and 1:
\texttt{? - rnd.real(5,:v)}
\begin{verbatim}
-> ( 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
\end{verbatim}

\texttt{rnd.rsnd}

\texttt{rnd.rsnd(number, number, | variable, number, number)}

This \textit{primitive} will unify or bind the third \textit{term} with a series of (floating point) random \textit{numbers} picked from a standard normal deviation where the first \textit{term} is the \textit{mean} and the second is the \textit{standard deviation}. The first \textit{term} is the count of random \textit{numbers} to be provided. For example:

\begin{verbatim}
? - 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
\end{verbatim}

\texttt{rnd.uint}

\texttt{rnd.uint(number, number| variable, number?, number?)}

This \textit{primitive} will unify or bind the second \textit{term} with a series of (unsigned integer) random \textit{numbers} picked in the range defined between the third and fourth \textit{terms}. The first \textit{term} is the count of random \textit{numbers} to be provided. For example:

\begin{verbatim}
? - 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
\end{verbatim}

When no range is provided, the random \textit{numbers} will all be in between 0 and the maximum value for a 64-bit unsigned integer:

\begin{verbatim}
? - 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
\end{verbatim}

\textbf{5.11 Range}

This section describes \textit{primitives} that handle \textit{ranges} or generate \textit{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.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:
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
   -> ( 0.10000 ) := 1.00 (0.002) 2
   -> ( 0.20000 ) := 1.00 (0.002) 3
   -> ( 0.30000 ) := 1.00 (0.003) 4
   -> ( 0.40000 ) := 1.00 (0.003) 5
   -> ( 0.50000 ) := 1.00 (0.004) 6
   -> ( 0.60000 ) := 1.00 (0.004) 7
   -> ( 0.70000 ) := 1.00 (0.005) 8
   -> ( 0.80000 ) := 1.00 (0.005) 9
   -> ( 0.90000 ) := 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 *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:

65
rng.uint(1,10,:x)

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

5.12 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",:1)
-> ( '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 3.1.6 on page 10.

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.
If a third term is provided, the primitive will unify it with all the matches 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

5.13 Symbol

This section describes primitives related to handling symbols.

**sym.cat**

\[
\text{sym.cat(term+, string|variable)}
\]

This primitive will unify or substitute 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

**sym.cmp**

\[
\text{sym.cmp(symbol, symbol, number|variable, symbol?)}
\]

The sym.cmp primitive will unify or substitute 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.sub**

\[
\text{sym.sub(symbol, number, number, symbol|variable)}
\]
The **sym.sub primitive** will unify or substitute 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

### 5.14 String

This section describes *primitives* related to handling *strings*.

**str.cat**

\[
\text{str.cat(} \text{term+}, \text{string})
\]

This *primitive* will unify or substitute 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**

\[
\text{str.cmp(} \text{string, string, number|variable, symbol?})
\]

The *str.cmp primitive* will unify or substitute 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.find**

\[
\text{str.find(} \text{string, string, number|variable})
\]

The *str.find primitive* will unify or substitute its third *term* with the offset (starting from 0) within its first *term* where the second *term* was find. If there is no occurrence 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 occurrences of the second *term* in the *string*:

```prolog
?- str.find("abcdefcc","c",:p)
-> (2) := 1.00 (0.000) 1
-> (6) := 1.00 (0.001) 2
-> (? ) := 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*.

```prolog
str.flip
```

**str.flip(string,string|variable)**

The *str.flip* *primitive* will unify or substitute its second *term* with a *string* containing the content of the first *term* inverted:

```prolog
?- str.flip("hello, world!",:s)
-> ("!dlrow ,olleh") := 1.00 (0.001) 1
```

```prolog
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:

```prolog
?- 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:

```prolog
?- str.head("hello world!","HELLO",insensitive)
-> ( ) := 1.00 (0.000) 1
```

```prolog
str.length
```

**str.length(string,number|variable)**

This *primitive* will unify or substitute its second *term* with the length (that is the number of characters) in the *string* passed as first *term*.

```prolog
?- 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 substitute 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 substitute 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 substitute its third term with its first term where all occurrences 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 if 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)

This *primitive* will unify or substitute 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

str.tolower

str.tolower(string,string|variable)

The *str.tolower* *primitive* will unify or substitute 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 substitute 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* version 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

\texttt{str.toupper(string,string|variable)}

The \texttt{str.toupper} primitive will unify or substitute its second term with a copy of first term where all alphabetic characters have been converted to uppercase:

\begin{verbatim}
?- str.toupper(\text{"HeLLo"},:s)
\rightarrow ( \text{"HELLO"} ) := 1.00 (0.000) 1
\end{verbatim}

str.tosym

\texttt{str.tosym(string,symbol|variable)}

The \texttt{str.tosym} primitive will unify or substitute its second term with symbol based on its first term. For example:

\begin{verbatim}
?- str.tosym(\text{"HeLLo"},:s)
\rightarrow ( \text{HeLLo} ) := 1.00 (0.000) 1
?- str.tosym(\text{"3.14"},:s)
\rightarrow ( \text{a3.14} ) := 1.00 (0.000) 1
?- str.tosym(\text{"hello, world."},:s)
\rightarrow ( \text{hello._world.} ) := 1.00 (0.000) 1
\end{verbatim}

str.trim

\texttt{str.trim(string,variable)}

\texttt{str.trim(string,variable,string)}

This primitive will unify or substitute its second term with its first term trimmed of any empty spaces at the start and end of the string. For example:

\begin{verbatim}
?- str.trim(\text{" this is my string ",:s})
\rightarrow ( \text{\"this is my string\"} ) := 1.00 (0.001) 1
\end{verbatim}

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:

\begin{verbatim}
?- str.trim(\text{"555-1234",:s,"555-"})
\rightarrow ( \text{"1234"} ) := 1.00 (0.000) 1
\end{verbatim}

str.trim.head

\texttt{str.trim.head(string,variable)}

\texttt{str.trim.head(string,variable,string)}

This primitive will unify or substitute 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:

\begin{verbatim}
?- str.trim.head(\text{" this is my string ",:s})
\rightarrow ( \text{\"this is my string \"} ) := 1.00 (0.001) 1
?- str.trim.head(\text{"555-1234-555",:s,"555"})
\rightarrow ( \text{"-1234-555"} ) := 1.00 (0.000) 1
\end{verbatim}
This *primitive* will unify or substitute 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:

```prolog
?- 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
```

### 5.15 Typing

This section describes *primitives* that can be used to check the type of any terms.

#### is.atom

The *primitive* will resolve to a truth value of 1 if the term is an atom, 0 otherwise. For example:

```prolog
?- 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

The *primitive* will resolve to a truth value of 1 if the term is a binary, 0 otherwise. For example:

```prolog
?- 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('aGVsbG8sIHdvcmxkIQA=')
   -> ( ) := 1.00 (0.000) 1
```

#### is.list

The *primitive* will resolve to a truth value of 1 if the term is a list, 0 otherwise. For example:
is.list

- is.list(34)
  -> ( ) := 0.00 (0.000) 1
- is.list([a,b,c,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.number

\textbf{is.number}(\textit{term})

The \textit{primitive} will resolve to a \textit{truth value} of 1 if the \textit{term} is a \textit{number}, 0 otherwise. For example:

?- \text{is.number}(3)
\rightarrow ( \ ): = 1.00 (0.055) 1
?- \text{is.number}(\text{hello})
\rightarrow ( \ ): = 0.00 (0.000) 1

is.odd

\textbf{is.odd}(\textit{number})

The \textit{primitive} will resolve to a \textit{truth value} of 1 if the \textit{term} is an odd \textit{number}, 0 otherwise. For example:

?- \text{is.odd}(3)
\rightarrow ( \ ): = 1.00 (0.000) 1
?- \text{is.odd}(4)
\rightarrow ( \ ): = 0.00 (0.000) 1

is.range

\textbf{is.range}(\textit{term})

The \textit{primitive} will resolve to a \textit{truth value} of 1 if the \textit{term} is a \textit{range}, 0 otherwise. For example:

?- \text{is.range}(<1|10>)
\rightarrow ( \ ): = 1.00 (0.000) 1
?- \text{is.range}(231)
\rightarrow ( \ ): = 0.00 (0.000) 1

is.regexp

\textbf{is.regexp}(\textit{term})

The \textit{primitive} will resolve to a \textit{truth value} of 1 if the \textit{term} is a \textit{regexp}, 0 otherwise. For example:

?- \text{is.regexp}(42)
\rightarrow ( \ ): = 0.00 (0.001) 1
?- \text{is.regexp}(\text{regexp("\d*"))}
\rightarrow ( \ ): = 1.00 (0.000) 1

is.string

\textbf{is.string}(\textit{term})

The \textit{primitive} will resolve to a \textit{truth value} of 1 if the \textit{term} is a \textit{string}, 0 otherwise. For example:

?- \text{is.string}(3)
\rightarrow ( \ ): = 0.00 (0.000) 1
?- \text{is.string}(\text{hello})
\rightarrow ( \ ): = 0.00 (0.001) 1
?- \text{is.string}(\text{"hello, world!"})
\rightarrow ( \ ): = 1.00 (0.000) 1
is.symbol

\texttt{is.symbol(term)}

The \textit{primitive} will resolve to a \textit{truth value} of 1 if the \textit{term} is a \textit{symbol}, 0 otherwise. For example:

\begin{verbatim}
?- is.symbol(3)
\rightarrow (  ) := 0.00 (0.000) 1
?- is.symbol(hello)
\rightarrow (  ) := 1.00 (0.000) 1
?- is.symbol("hello, world!")
\rightarrow (  ) := 0.00 (0.000) 1
\end{verbatim}

is.variable

\texttt{is.variable(term)}

The \textit{primitive} will resolve to a \textit{truth value} of 1 if the \textit{term} is an unbound \textit{variable}, 0 otherwise. For example:

\begin{verbatim}
?- is.variable(:h)
\rightarrow ( :h ) := 1.00 (0.000) 1
?- is.variable(5)
\rightarrow (  ) := 0.00 (0.000) 1
?- set(:h,5), !is.variable(:h)
\rightarrow ( 5 ) := 1.00 (0.000) 1
\end{verbatim}

6 Elementals

This section provides some details on all the \textit{elementals} supported by the \textit{runtime}. For each one, the list of supported \textit{properties} and accepted values will be given as well as some explanation on their use cases.

MRKCBFSolver

This \textit{elemental} class is the most common one used in \textit{fizz}. It is in fact the default and can handle \textit{statements} as well as \textit{prototypes}. It implement a \textit{breadth-first} solving which is optimized for concurrency, therefore it is not the most efficient \textit{solver} with regard to time and memory usage. However, at this moment, it is the only \textit{elemental} capable of handling \textit{prototypes}.

This \textit{elemental} supports the following \textit{properties}:

\begin{itemize}
\item \texttt{p.limit} the maximum number of \textit{prototype} the object will accept when they are defined.
\item \texttt{s.limit} the maximum number of \textit{statement} the object will accept when they are asserted.
\item \texttt{replies.are.triggers} set to \texttt{no} to instruct the \textit{elemental} to not considere \textit{replies} as potential triggers.
\item \texttt{memoize} set to \texttt{yes} to instruct the \textit{elemental} to use memoization (that is to temporary cache replies to queries in order to avoid inferring the same thing multiple time).
\end{itemize}

When such \textit{elemental} is set to memoize, cached \textit{statements} will be periodically cleared at a a frequency set by the \texttt{mzttl substrate} configuration.
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 retrieve 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.

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.

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 column is to be ignored.
- **select** the column 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:

```plaintext
1 car { 
2   class = MRKCCSVStore, 
3   filepath = "./etc/data/cars.csv", 
4   delimiter = ":", 
5   offset = 2, 
6   no.match = fail, 
7   offloaded = yes 
8 } {} 
```

MRKCSBFStore

This elemental provides a way to store and retrieve statements from a binary file. While it is a slower way to retrieve 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 retrieval 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.

**FZZCLGProcessor**

This *elemental* provides an interface to the *Link Grammar Parser*\(^4\) 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 be made available via a series of lists. In order to be able to use this *elemental*, the module in which it resides (modLGR) must be loaded in *fizz* using either the /use command or a solution file.

This *elemental* supports the following *properties*:

---

\(^4\)http://www.link.cs.cmu.edu/link/
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:

```fizz
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 the string to be parsed followed by four unbound variables:

```fizz
?- #lgr.parse("the quick brown fox jumps over the lazy dog.",:ws,:ls,:ln,:cn)
\rightarrow ( [[[], nil], [[], "the"], [[a], "quick"], [[a], "brown"], [[n], "fox"], [[v], "jumps"], [[], ",over"], [[], "the"], [[a], "lazy"], [[n], "dog"], [[], ",."], [[], nil]], [[X, [p], 0, 10], [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 (0.021) 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 be unified to a tree which describes the components in the sentence as generated by the Phrase Parser.

We will now define the contents of each of the list, starting with the words list:

```fizz
?- #lgr.parse("the quick brown fox jumps over the lazy dog.",:ws,:,ls,,_)
\rightarrow ( [[[], 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.

```fizz
?- #lgr.parse("the quick brown fox jumps over the lazy dog.",:ws,,ls,,_)
\rightarrow ( [[X, [p], 0, 10], [MV, [], 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
```

5http://www.link.cs.cmu.edu/link/ph-explanation.html

6see section 3.3 in https://www.abisource.com/projects/link-grammar/dict/introduction.html for a list of the subscripts
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\(^7\), 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

The 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.

FZZCWebAPIGetter

The FZZCWebAPIGetter elemental performs a 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:

<table>
<thead>
<tr>
<th>property</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>headers</td>
<td>an optional frame describing all the headers to be added to the request</td>
</tr>
<tr>
<td>flags</td>
<td>a set of symbols modifying the behavior of the JSON to frame converter. The flag <code>stringify</code> will keep all strings as string terms. <code>symbolize</code> will force all strings to be converted as symbols. The default behavior is to convert the strings that can be considered symbol as such</td>
</tr>
<tr>
<td>url.host</td>
<td>the scheme and hostname of the web service (http or https)</td>
</tr>
<tr>
<td>url.path</td>
<td>the path of the requested resource</td>
</tr>
<tr>
<td>verbose</td>
<td>an optional boolean value (or a symbol <code>true</code>, <code>false</code>) to instructs the elemental to output more traces in the console</td>
</tr>
</tbody>
</table>

For example, to get any conversion rate from `api.fixer.io`, we would define the elemental as follow:

```lisp
1  fixer.get {  
2  3    class     = FZZCWebAPIGetter,  
4    url.host   = "http://api.fixer.io",  
5    url.path   = "/latest",  
6  7 } {  
8 9 }
```

\(^7\)see https://www.abisource.com/projects/link-grammar/dict/index.html for details
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",
bose = "noreferrer", Server = "cloudflare", CF-RAY = "4168159a27192f4-5JC"}, {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 = 1076.400000, 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 = 8.328000, 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 contain 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 retrieved. 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.

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,
```
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 = 7.782300, 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}})
```

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 reuires various values to be provided in its properties. The following table contains them:

<table>
<thead>
<tr>
<th>property</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>query</td>
<td>the predicate (in the form of a functor) to be used to query for statements</td>
</tr>
<tr>
<td>generalize</td>
<td>a list of lists describing which of the statements terms will be considered</td>
</tr>
<tr>
<td>formatting</td>
<td>a list describing how each of the terms in a statement is to be understood</td>
</tr>
<tr>
<td>hidden_layers</td>
<td>a number providing the number of hidden layers to be used by the neural</td>
</tr>
<tr>
<td>neurons_in_hidden_layers</td>
<td>a number providing the number of neurons in each hidden layers.</td>
</tr>
</tbody>
</table>

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 = FZZCFFBNNetwork,
    alias = iris.ffbn,
    query = iris(_,_,_,_,_),
```
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*.

**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:

```prolog
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:

```prolog
?- /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:

```prolog
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:

```prolog
?- /spy(append,rand,dad)
spy : observing rand
spy : observing dad
```
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:

```prolog
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:

```prolog
?- /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(1630) := 1.00
spy : S dad(charles1) := 1.00
spy : S rand(1633) := 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
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(charles2) := 1.00
```
**FZZCTicker**

This *elemental* can be used to activate other *elemental* at a regular interval by firing a *statement*. For example:

```plaintext
1 tick {
2    class = FZZCTicker,
3    mod  = 4
4 } {
5 }
6 }
```

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*:

```plaintext
?- /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:

```plaintext
1 tick {
2    class = FZZCTicker,
3    tick  = 1.5
4 } {
5 }
6 }
```

**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 retrieval of *statements* (see the sample *cars.fizz* for an example).

**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 storage.

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 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:

```prolog
1 assert.stamp {  
2   (:f, :v) :- assert(:f, :v, {stamp = %now});  
3 }  
```

To create it from the console, we would type this:

```prolog
?- 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. An *escape sequence* only works for a single substitution, therefore, we could have used multiple \, one for each level of depth to protect the *term* for. 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*.
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 substituted 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 `set` to assign the escaper 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:

```erlang
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 substitute 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. The service will only returns what was collected once the timeout occurs.

As an example, let’s consider the following knowledges:

```erlang
1 product {
2   (model_e,tesla,2012);
3}
```
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, 2007) := 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
```

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 is half a second.

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 5.5 on page 50) 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

0.5.0-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 7)
  - noisy (see section 2.5 on page 7)
  - clone (see section 2.5 on page 7)
- 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 86)
- primitives gt, gte, lt and lte now also works with strings and symbols

Additions

- solution files (see section 4.3 on page 19)
- new console command: /use (see section 4.4 on page 32)
- 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 3.1.6 on page 10)
- new primitives:
  - frm.erase (see section 5.4 on page 46)
  - lst.mix (see section 5.6 on page 54)
  - lst.sort (see section 5.6 on page 55)
  - lst.sub (see section 5.6 on page 56)
– **rex.make** (see section 5.12 on page 66)
– **rex.match** (see section 5.12 on page 66)
– **rng.rand** (see section 5.11 on page 66)

*new constraints:*

– eq
– is.regexp
– is.bound

*new volatiles: sym.3, sym.4 and sym.10* (see section 3.8 on page 15)

**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 77)
  - MRKCCSVStore (see section 6 on page 77)
  - FZZCLGRProcessor (see section 6 on page 78)

- new terms:
  - guid (see section 3.1.5 on page 10)

- new primitives:
  - str.trim.head (see section 5.14 on page 72)
  - str.trim.tail (see section 5.14 on page 73)
  - str.tail (see section 5.14 on page 70)
  - str.head (see section 5.14 on page 69)
  - lst.incl (see section 5.6 on page 52)
  - lst.excl (see section 5.6 on page 51)
  - lst.join (see section 5.6 on page 53)
  - lst.init (see section 5.6 on page 52)
  - sym.cmp (see section 5.13 on page 67)
  - sim (see section 5.1 on page 35)
  - is.even (see section 5.15 on page 74)
  - is.odd (see section 5.15 on page 75)
  - gid.make (see section 5.9 on page 62)

- 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 occurring
- `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 44)
  – bundle (see section 5.2 on page 37)
  – div.int (see section 5.1 on page 34)
  – fzz.lst (see section 5.9 on page 61)
  – lst.remove (see section 5.6 on page 54)
  – mao.sign (see section 5.8 on page 61)
  – str.find (see section 5.14 on page 68)
  – str.flip (see section 5.14 on page 69)
  – str.trim (see section 5.14 on page 72)
  – str.rest (see section 5.14 on page 70)
  – str.swap (see section 5.14 on page 70)
  – sym.cat (see section 5.13 on page 67)
• new console commands:
  – /reload (see section 4.4 on page 29)
  – /import.txt (see section 4.4 on page 27)
• new class FZZCWebAPIGetter (see section 6 on page 80)

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 26 and 4.4 on page 23)
- added primitive `change` (see section 5.2 on page 38)
- added primitive `console.exec` (see section 5.2 on page 38)
- added primitive `then` (see section 5.2 on page 43)
- added primitive `tme.str` (see section 5.2 on page 43)
- added primitive `str.cmp` (see section 5.14 on page 68)
- added elemental class `FZZCWebAPIPuller` for fetching JSON data from web services (see section 6 on page 81)

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 86)

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

Concepts
Elemental, 7
Knowledge, 2
Predicate, 4
Prototype, 6
Service, 8
Statement, 3

Console
bye, 20
cpus, 21
create, 21
delete, 21
export.csv, 21
export.json, 23
freeze, 24
history.cls, 24
history.len, 25
import.csv, 25
import.json, 26
import.txt, 27
kindle, 28
knows, 28
list, 29
load, 29
peek, 33
poke, 30
reload, 29
save, 30
scan, 30
spy, 31
stats, 31
tells, 32
unload, 32
use, 32
wipe, 33

Elementals
FZZCFFBNetwork, 82
FZZCLGRProcessor, 78
FZZCRandomizer, 83
FZZCTicker, 86
FZZCWebAPIGetter, 80
FZZCWebAPIPuller, 81
MRKC BFSolver, 76
MRKCCSVStore, 77
MRKCLettered, 86
MRKCSBFStore, 77

Miscellaneous
Escaper, 87

Primitives

Arithmetic
add, 33
div.int, 34
div, 33
inv, 34
mod, 34
mul, 35
sim, 35
sub, 35
sum, 36

Basic
assert, 36
break:not, 37
break, 37
bundle, 37
change, 38
console.exec, 38
console.gets, 38
console.puts, 38
declare, 38
define, 39
false, 40
forget, 40
fuzz, 40
hush, 40
now, 40
peek, 41
poke, 41
repeal, 42
revoke, 42
set:if:not, 43
set:if, 43
set, 42
then, 43
tme.str, 43
true, 44
whisper, 44

Boolean Logic
boo:and, 57
boo:not, 57
boo:or, 57
boo:xor, 57

Comparaisons
aeq, 44
are:different, 44
are:same, 45
cmp, 45
eq, 45
gte, 45
gt, 45
lte, 46
lt, 45
neq, 46
Frame
  frm.cat, 49
  frm.empty, 47
  frm.erase, 46
  frm.fetch, 46
  frm.labels, 48
  frm.label, 47
  frm.length, 47
  frm.make, 47
  frm.pairs, 49
  frm.store, 47
  frm.sub, 49
  frm.values, 48
Functor
  fun.label, 50
  fun.length, 49
  fun.make, 50
  fun.member, 50
  fun.terms, 50
List
  lst.cat, 51
  lst.diff, 51
  lst.empty, 51
  lst.except, 51
  lst.excl, 51
  lst.flip, 52
  lst.head, 52
  lst.incl, 52
  lst.init, 52
  lst.item, 53
  lst.join, 53
  lst.length, 53
  lst.make, 54
  lst.member, 54
  lst.mix, 54
  lst.remove, 54
  lst.rest, 55
  lst.sort, 55
  lst.span, 55
  lst.sub, 56
  lst.swap, 56
  lst.tail, 56
Mathematics
  mao.abs, 58
  mao.ceil, 58
  mao.exp, 58
  mao.floor, 59
  mao.log10, 59
  mao.log, 59
  mao.modf, 60
  mao.pow, 60
  mao.round, 60
  mao.sign, 61
  mao.sqrt, 61
Miscellaneous
  fzz.lst, 61
  gid.make, 62
  gid.str, 62
  gid.sym, 62
Random
  rnd.real, 62
  rnd.rsnd, 63
  rnd.uint, 63
Range
  rng.clamp, 64
  rng.inc, 64
  rng.inter, 64
  rng.max, 64
  rng.min, 64
  rng.rand, 66
  rng.span, 65
  rng.uint, 65
  rng.union, 65
Regexp
  rex.make, 66
  rex.match, 66
String
  str.cat, 68
  str.cmp, 68
  str.find, 68
  str.flip, 69
  str.head, 69
  str.length, 69
  str.rest, 70
  str.sub, 70
  str.swap, 70
  str.tail, 70
  str.tokenize, 71
  str.tolower, 71
  str.tonum, 71
  str.tosym, 72
  str.toupper, 72
  str.trim.head, 72
  str.trim.tail, 73
  str.trim, 72
  sym.cat, 67
Symbol
  sym.cmp, 67
  sym.sub, 67
Typing
  is.atom, 73
  is.binary, 73
  is.even, 74
  is.final, 74
  is.frame, 74
  is.func, 74
is.list, 73
is.number, 75
is.odd, 75
is.range, 75
is.regexp, 75
is.string, 75
is.symbol, 76
is.variable, 76

Services
    MRKCCollector, 88
    MRKCEvaluator, 89