Building a simple stock market monitor with *fizz*

Jean-Louis Villecroze
jlv@f1zz.org @CocoaGeek

May 24, 2018

Abstract

In this article[1], we will construct a simple (and very minimally *intelligent*) application which will monitor the changes happening on a set of stock prices. We will also look at how to integrate some *Machine Learning* into the application. The complete solution can be found in the `etc/articles/iex` folder of *fizz* 's distribution.

What is *fizz* ?

fizz is an experimental language and runtime environment for the exploration of *cognitive architectures* and software solutions combining *Machine Learning* (ML) and *Machine Reasoning* (MR). It is based primarily on *symbolic logic programming* and *fuzzy formal logic*, and it features a distributed, concurrent, asynchronous and responsive *inference engine* as well as a built-in *neural network* implementation. If you have dabbled in the past with *PROLOG*, then you will feel some familiarities as *fizz* shares some of its concepts and syntax with it. It is important, however, to keep in mind that *fizz* is not *PROLOG*.

Since it's tradional to introduce a programming language with the famous *hello, world!* example, here's the *fizz* version:

```
hello {

    () :- console.puts("Hello, World!");

}
```

We will then have to invoke it from the console:

```
$ ./fizz.x64 hello.fizz
fizz 0.3.0-X (20180519.2228) [x64|8|w|l]

load : loading hello.fizz ...
load : loaded  hello.fizz in 0.001s
load : loading completed in 0.001s
?- #hello
Hello, World!
-> (  ) := 1.00 (0.001) 1
```

Before continuing further with *fizz* , we need to define some of the terminology which we will encountering in this article:

Substrate	is the *runtime environment* provided by *fizz* .
Knowledge	is a collection of related *statements* and/or *prototypes*.
Statement	is a collection of *terms* with an assigned *truth value* (a.k.a. facts).
Predicate	is a labeled collection of *terms* with an assigned *truth value* range (or *variable*) that are used to *query* the *substrate*.
Prototype	is a chained collection of *predicates* with an *entry-point* that can be inferred upon (a.k.a. rules).
Elemental	is a runtime object (that lives in the *substrate*) which hold *knowledge* and can return answers to queries.
Service	is a runtime object (that lives in the *substrate*) which provide a unique service within the *runtime*.

One of the core concepts that set *fizz* aside from tradional *PROLOG* implementations, 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 object (*elementals*) each having access only to what they must know (*knowledges*). *Elementals* in *fizz* are

[1]Thanks to Robert Wasmann (@retrospasm) for providing feedback and reviewing this document.

independent *actors*, which must exchange messages (by using a queries and replies mechanism) in order to execute any inferences. While this is far from being the most efficient method (and performance does suffer) it allows for a system to be build to be responsive. A *statement* that is broadcasted in a *substrate* will potentially trigger the execution of any *prototype* that references it. This strict isolation between *elementals* supports inferences to be continued at a later time (within reason) as new data becomes available. This also supports inferences to be distributed among many cores and (eventually) many participating hosts.

Statements, and *predicates* are composed of collection of data items which are called *terms*. In *fizz* , there is a total of eight such different types of *terms*:

Atom	is the representation of an atomic piece of data, such as a number, a symbol or a string.
Constant	is a special kind of *variable* which holds a static value.
Frame	is a dictionary which stores key-value pairs.
Functor	is a named *list*.
List	is a read-only collection of *terms*.
Range	is the expression of a *range* of numerical values between *minimum* and *maximum* values.
Variable	is a placeholder for any *term*.
Volatile	is a special kind of *variable* which holds an ever changing value.

To allow for complex solutions to be built, *fizz* allows for the *class* of an *elemental* to be specified. While in general, most *elementals* are just instances of the same underlying base *class* (in which case there is no need to specify a class when defining the *elemental*), often it is necessary to use one of the classes provided by the *runtime*. This is something that we will do many times in this article.

For further details on *fizz* , including small examples of its syntax and capabilities, please refer to the *user manual*. Let's now get on with building this application ...

Fetching data from the *web API*

The first step we are going to take in building this application is to look at fetching the stock prices from the *web API* we have decided to use (IEX Trading[2]). *fizz* provides the `FZZCWebAPIGetter` class of *elemental* which performs this operation. Since that *elemental* only fetches data when queried (unlike the class `MRKCWebAPIPuller`), we are going to need to complement it with an `FZZCTicker` *elemental*, so that we can query for the latest stock prices at a regular interval. The advantage of using this *elemental* is that we can more easily dynamically change the query used to request data from the web service.

To get started, let's create a new *fizz* source file which we will call `iex.core.fizz` and setup the requiered *knowledge* definition for the two *elementals* we just discussed:

```
iex.tick {
    class          = FZZCTicker,
    tick           = 5,
    tick.on.attach  = yes
} {}

iex.get {
    class    = FZZCWebAPIGetter,
    url.host = "https://api.iextrading.com",
    url.path = "/1.0/stock/market/batch"
} {}
```

Whenever the `iex.tick` *elemental* ticks, it will fire a *statement* (every five seconds and at the launch of the system, as we have specified `yes` for the property `tick.on.attach`). As we want to trigger `iex.get` into fetching data from the *web API* we have described (with the `url.host` and `url.path` properties), we need now to introduce a third *elemental* which when triggered by `iex.tick` will query `iex.get` and thus get it to fetch data from the *web API*:

[2] Data provided for free by IEX

```
1 iex.query {
2
3     () :-   @iex.tick(_,_),
4             #iex.get({types=quote,symbols=[AAPL],filter=[latestPrice,change,latestUpdate]},[:t,200,_,:c]),
5             console.puts(:t," ",:c),
6             hush;
7
8 }
```

In it, we defined a single *prototype* which in *line 3* specifies `iex.tick` as a triggering *predicate* (we won't care about its *terms* so we used *wildcard variables* which will unify with anything), and query `iex.get` for the `AAPL` (*Apple Inc.*) quote data. The first *term* of a *predicate* querying an `FZZCWebAPIGetter` *elemental* is always a *frame* describing the query part of a *web API*'s request to be performed (and as such, it will always be specific to the *web API* you are using). In return, the *elemental* will answer with a *statement* where the second *term* is unified with a list containing: a timestamp, an HTTP status code, a frame containing the HTTP response's headers and finally the *frame* containing the received content. In *line 4*, we unify the *term* directly to a *list* so that we can rely on the unification process to extract the *terms* we care about and also insure that the *predicate* will only be satisfied if the HTTP status code is `200`. For now, we will just output the received data to the console (with the *primitive* `console.puts`) and since it doesn't make sense for this *elemental* to publish the *statements* it generates by successfuly evaluating the *prototype*, we call the `hush` *primitive* to turn the evaluation silent.

The above example shows one predicate starting with a `@` and another one starting with a `#`. The former indicates to *fizz* that the `iex.tick` predicate is to be considered a *triggering predicate* while the latter, used with `iex.get` is a standard non-primitive *predicate*. In theory, all *predicates* to *knowledge* could be *triggering* but that will not be practical, as a single *query* may cause a long cascade of *inferencing*. Therefore, it is something that must be specified.

Let's now launch *fizz* and load `iex.core.fizz` and see that we are getting quotes every 5 seconds (we will later change this to 10 seconds):

```
$ ./fizz.x64 iex.core.fizz
fizz 0.3.0-X (20180519.2228) [x64|8|w|l]

load : loading iex.core.fizz ...
load : loaded  iex.core.fizz in 0.004s
load : loading completed in 0.005s
1525807078.352411 {AAPL = {quote = {latestPrice = 185.520000, change = 0.360000, latestUpdate = 1525807077067}}}
1525807083.910785 {AAPL = {quote = {latestPrice = 185.520000, change = 0.360000, latestUpdate = 1525807077067}}}
1525807088.292208 {AAPL = {quote = {latestPrice = 185.520000, change = 0.360000, latestUpdate = 1525807077067}}}
1525807093.201322 {AAPL = {quote = {latestPrice = 185.520000, change = 0.360000, latestUpdate = 1525807077067}}}
1525807098.198470 {AAPL = {quote = {latestPrice = 185.580000, change = 0.420000, latestUpdate = 1525807094800}}}
1525807103.228156 {AAPL = {quote = {latestPrice = 185.590000, change = 0.430000, latestUpdate = 1525807100087}}}
```

To make things more modular and facilitate further extensions of the application, we are going now to turn the `symbols` and `filter` *lists* expected by the *web API* in independant *factual knowledge* that will get queried each time we want a request to be sent to the *web service*. We will also sieze the opportunity to specify two other stock tickers we are interested in. Create a new *fizz* source file called `iex.vars.fizz` and add the following two *knowledge definitions* to it:

```
iex.symbols {

    ([AAPL,GOOGL,MSFT]);

}

iex.filters {

    ([latestPrice,change,changePercent,latestUpdate]);

}
```

Both declare a single *statement* that will be queried by the modified *prototype* of the `iex.query` *elemental* in order to retrieve the *lists*. Here's the new version of `iex.query`:

```
1  iex.query {
2
3      () :-   @iex.tick(_,_),
4              #iex.symbols(:s),
5              #iex.filters(:f),
6              #iex.get({types=quote,symbols=:s,filter=:f},[:t,200,_,:c]),
7              console.puts(:t," ",:c),
8              hush;
9
10 }
```

In *line 4 and 5* we fetch both *lists* and bound them to the *variables* `s` and `f`. We then use these two *variables* to compose the query to be sent to the *web API*. The process by which *bounded variables* are replaced by their *bound values* is called *substitution*. If we reload *fizz* with the updated file, we can verify that we are now getting the three quotes we asked for:

```
$ ./fizz.x64 iex.core.fizz iex.vars.fizz
fizz 0.3.0-X (20180519.2228) [x64|8|w|l]

load : loading iex.core.fizz ...
load : loading iex.vars.fizz ...
load : loaded  iex.vars.fizz in 0.003s
load : loaded  iex.core.fizz in 0.016s
load : loading completed in 0.016s
1525807526.571183 {AAPL = {quote = {latestPrice = 185.500000, change = 0.340000, changePercent = 0.001840, latestUpdate =
     1525807525584}}, GOOGL = {quote = {latestPrice = 1059.200000, change = -0.260000, changePercent = -0.000250,
     latestUpdate = 1525807469386}}, MSFT = {quote = {latestPrice = 95.600000, change = -0.620000, changePercent = -0.006440,
      latestUpdate = 1525807523001}}}
1525807531.511069 {AAPL = {quote = {latestPrice = 185.500000, change = 0.340000, changePercent = 0.001840, latestUpdate =
     1525807525584}}, GOOGL = {quote = {latestPrice = 1059.200000, change = -0.260000, changePercent = -0.000250,
     latestUpdate = 1525807469386}}, MSFT = {quote = {latestPrice = 95.610000, change = -0.610000, changePercent = -0.006340,
      latestUpdate = 1525807527562}}}
```

From JSON data to factual knowledge

Now that we are getting the data from the *web API*, we are going to look at turning them into *factual knowledge* which can then be used in any sort of *logical inferencing*. For that, we are going to rely on an *elemental* using *procedural knowledge* to process each set of quote data we get. But first, we need to modify `iex.query` to break down the content of the root *frame* we are retrieving from the *web API*. For that, we will use the *primitive* `frm.fetch`:

```
1  iex.query {
2
3      () :-   @iex.tick(_,_),
4              #iex.symbols(:s),
5              #iex.filters(:f),
6              #iex.get({types=quote,symbols=:s,filter=:f},[:t,200,_,:c]),
7              frm.fetch(:c,:l,:d),
8              console.puts(:t," ",:l," ",:d),
9              hush;
10
11 }
```

On *line 7*, we provide to the *primitive* the value of the *variable* `c` which is unified on *line 6* with the *frame* containing the content received from the *web service*. Since the second and third *terms* are *unbound variables*, the *primitive* will generate a *statement* for each of the key-value *pairs* in the *frame*. Each of these *statements* will then be considered *concurrently* by the *solver* for the rest of the *prototype* execution. We can test this if we save the modified file and start *fizz* again:

```
$ ./fizz.x64 iex.core.fizz iex.vars.fizz
fizz 0.3.0-X (20180519.2228) [x64|8|w|l]

load : loading iex.core.fizz ...
load : loading iex.vars.fizz ...
load : loaded  iex.vars.fizz in 0.003s
load : loaded  iex.core.fizz in 0.016s
load : loading completed in 0.016s
1525837383.118558 AAPL {quote = {latestPrice = 186.050000, change = 0.890000, changePercent = 0.004810, latestUpdate =
      1525809600267}}
1525837383.118558 GOOGL {quote = {latestPrice = 1058.590000, change = -0.870000, changePercent = -0.000820, latestUpdate =
      1525809600267}}
1525837383.118558 MSFT {quote = {latestPrice = 95.810000, change = -0.410000, changePercent = -0.004260, latestUpdate =
      1525809600220}}
```

We will now replace the call to the `console.puts` *primitive* by a *predicate* which will further process each of the *stock ticker*'s data. In general, it is recommended to break down inferencing over multiple *elementals* to take advantage of *fizz* 's concurrent nature. Since each of the relevant quotes is contained in a *frame* under the *key* `quote`, we will first extract the sub-*frame* by specifying it in the *prototype*'s *entry-point*:

```
iex.proc {

    (:t,:l,{quote = :d}) :- console.puts(:l," : ",:d),
                            hush;
}
```

In order for a *prototype* to be selected by the *solver*, its *entry-point* must successfuly unify with the *predicate* that is under consideration. That process will insure that the *prototype* will be executed only if the third *term* is a *frame* which contains a value for the *key* `quote`. This value will be bound to the *variable* `d`. For now we will just print it to the console.

We now modify `iex.query` as follows to query `iex.proc` with each of the fetched *frames*:

```
iex.query {

    () :-   @iex.tick(_,_),
            #iex.symbols(:s),
            #iex.filters(:f),
            #iex.get({types=quote,symbols=:s,filter=:f},[:t,200,_,:c]),
            frm.fetch(:c,:l,:d),
            #iex.proc(:t,:l,:d),
            hush;

}
```

Let's run the modified *knowledge*:

```
$ ./fizz.x64 iex.core.fizz iex.vars.fizz
fizz 0.3.0-X (20180519.2228) [x64|8|w|l]

load : loading iex.core.fizz ...
load : loading iex.vars.fizz ...
load : loaded  iex.vars.fizz in 0.003s
load : loaded  iex.core.fizz in 0.016s
load : loading completed in 0.016s
AAPL : {latestPrice = 186.050000, change = 0.890000, changePercent = 0.004810, latestUpdate = 1525809600267}
GOOGL : {latestPrice = 1058.590000, change = -0.870000, changePercent = -0.000820, latestUpdate = 1525809600267}
MSFT : {latestPrice = 95.810000, change = -0.410000, changePercent = -0.004260, latestUpdate = 1525809600220}
```

The next step we are going to look at is the transformation of the *frame* into a *statement*. We will accomplish that with the primitives `frm.fetch` and `assert`. The latter allows for a *statement* constructed from a *functor* to be declared and stored within the *substrate*. Here's the modified `iex.proc` definition:

```
1 iex.proc {
2
3     (:t,:l,{quote = :d}) :- frm.fetch(:d,latestUpdate,:u), div(:u,1000,:rt),
4                             frm.fetch(:d,latestPrice,:p),
5                             frm.fetch(:d,change,:c),
6                             frm.fetch(:d,changePercent,:cp),
7                             assert(iex.quote.data(:l,:rt,:p,:c,:cp)),
8                             hush;
9 }
```

Since the value we get for `latestUpdate` needs to be divided by a thousand, we do so in *line 3* using the `div` *primitive*. Once we have fetched the 4 values from the *frame* bounded to the *variable* `d`, we use `assert` to create a new statement with the label `iex.quote.data` passing in it the stock ticker's symbol, the timestamp of the price update as well as the change value and percent of change (both of which are computed from the price at the last closing). If we now run*fizz* again, we can observe the *statements* as they get generated:

```
$ ./fizz.x64 iex.core.fizz iex.vars.fizz
fizz 0.3.0-X (20180519.2228) [x64|8|w|l]

load : loading iex.core.fizz ...
load : loading iex.vars.fizz ...
load : loaded  iex.vars.fizz in 0.003s
load : loaded  iex.core.fizz in 0.016s
load : loading completed in 0.016s
?- /spy(append,iex.quote.data)
spy : observing iex.quote.data
spy : S iex.quote.data(AAPL, 1525809600.267000, 186.050000, 0.890000, 0.004810) := 1.00 (100.000000)
spy : S iex.quote.data(GOOGL, 1525809600.267000, 1058.590000, -0.870000, -0.000820) := 1.00 (100.000000)
spy : S iex.quote.data(MSFT, 1525809600.220000, 95.810000, -0.410000, -0.004260) := 1.00 (100.000000)
spy : S iex.quote.data(AAPL, 1525809600.267000, 186.050000, 0.890000, 0.004810) := 1.00 (100.000000)
spy : S iex.quote.data(MSFT, 1525809600.220000, 95.810000, -0.410000, -0.004260) := 1.00 (100.000000)
spy : S iex.quote.data(GOOGL, 1525809600.267000, 1058.590000, -0.870000, -0.000820) := 1.00 (100.000000)
```

In order to see the *statements* we have used the *console command /spy* to observe anything happening for the label `iex.quote.data` within the *substrate*. Take note of that *command* as it comes in handy when debugging ...

As we can see in the the above example, we are asserting the exact same *statement* as we may be fetching the data from the *web API* more often than they get updated (this will also be the case when stock market is closed). While *fizz* can avoid having multiple copies of the same *statement* in the *substrate*, there's a runtime cost associated with the assertions which we may want to avoid. At the same time, we will want to be able to query the very last price for a given stock ticker. The easiest way to do this will be to store in a *factual knowledge* the timestamp (the `latestUpdate` value we retrieved earlier) of each stock ticker we care about. Once we have the timestamp, we can use that *knowledge* in `iex.proc` to decide if the quote data we are processing is more recent than what we received last, and thus avoid asserting it again.

To implement this, we are going to start a new *fizz* source file which we will call `iex.data.fizz`. We will also use it store the quotes we will be receiving and later we will save the data into it. We write the following two *elemental* definitions in it:

```
iex.quote.last {
    class    = MRKCLettered,
    no.match = fail
} {}

iex.quote.data {
    class = MRKCLettered
} {}
```

Since both will only be containing *factual knowledge*, they are based on the `MRKCLettered` class of *elemental*. For `iex.quote.last`, we specify the *property* `no.match` with the value of `fail` to force the *elemental* to answer a query for which it doesn't have a successful answer by a fail instead of staying silent. Since inferencing in *fizz* can be faced with multiple instances of an `iex.quote.last` *elemental* (same or different *substrate*), the default behavior for such class of *elemental* is to stay silent when a query cannot be unified to any existing *statement*, as other *elementals* may be able to unify successfuly. Afterall, a lack of information does not necessary means that a *predicate* is false. In this example, we do need `iex.quote.last`, for which there will only be a single instance, to let us know when no match was found as it will always be the case for the first timestamp of a stock ticker.

We are now ready to modify `iex.proc` to add a `iex.quote.last` *predicate* in order to filter out the already received data. Once we really have new data, we will be replacing the previously asserted *statement* for the stock ticker by a new one with the new timestamp. For this, *fizz* provides a *primitive* called `change`:

```
1  iex.proc {
2
3      (:t,:l,{quote = :d}) :- frm.fetch(:d,latestUpdate,:u), div(:u,1000,:rt),
4                              !#iex.quote.last(:l,:rt),
5                              change([iex.quote.last(:l,_)],[iex.quote.last(:l,:rt)]),
6                              frm.fetch(:d,latestPrice,:p),
7                              frm.fetch(:d,change,:c),
8                              frm.fetch(:d,changePercent,:cp),
9                              #assert(iex.quote.data(:l,:rt,:p,:c,:cp)),
10                             hush;
11 }
```

Line 4 and *5* are the additions we made. We use `!` in front of the `iex.quote.last` *predicate*, to indicate that we are only interested by the failure to unify any of the *statements* stored in that *knowledge*. When the *predicate* is satisfied, that is when the last timestamp is different (or if this is the first data we are receiving for the particular ticker), the solver will continue onto the *primitive* `change`, which will request from the *substrate* to get the *statement* described (as a *functor*) in the first *list* replaced by the a *statement* build from the *functor* given in the second *list*. Since we do not, here, know the value of the last timestamp for the stock ticker, we use a *wildcard variable* to insure that whatever it was, the *statement* will be removed and replaced by the one with the new *timestamp*. Please note that the *primitive* `change` is asynchronous, meaning that the replacement of the *statement* will most likely not have been executed when the following *predicates* get evaluated. Depending on what the *prototype* is doing, this may be an issue.

To get a sense of what's going on when the application will be running, without having to use the `/spy` *console command*, we also changed the *predicate* to the *primitive* `assert` to a *knowledge* based *predicate* (by prefixing it with `#`). We can then add the following *procedural knowledge* to the file `iex.core.fizz`:

```
assert {

    (:f) :- console.puts("assert: ",:f), assert(:f);

}
```

This provides us with an easy way to output every *statement* that gets asserted at *runtime*. Let's try this out now:

```
$ ./fizz.x64 iex.core.fizz iex.vars.fizz iex.data.fizz
fizz 0.3.0-X (20180519.2228) [x64|8|w|l]

load : loading iex.core.fizz ...
load : loading iex.vars.fizz ...
load : loaded  iex.vars.fizz in 0.003s
load : loading iex.data.fizz ...
load : loaded  iex.data.fizz in 0.005s
load : loaded  iex.core.fizz in 0.012s
load : loading completed in 0.012s
assert: iex.quote.data(AAPL, 1525896000.459000, 187.360000, 1.310000, 0.007040)
assert: iex.quote.data(GOOGL, 1525896000.446000, 1088.950000, 30.360000, 0.028680)
```

```
assert: iex.quote.data(MSFT, 1525896000.207000, 96.940000, 1.130000, 0.011790)
?- #iex.quote.data(AAPL,:t,:p,_,_)
-> ( 1525896000.459000 , 187.360000 ) := 1.00 (0.001) 1
?- #iex.quote.last(AAPL,:t)
-> ( 1525896000.459000 ) := 1.00 (0.001) 1
```

You will note that this time around, we only saw a single assert (even though we let the application run for 15 seconds) for each of the stock tickers since this was captured past 11:00PM PST. And just to verify, we also queried the `iex.quote.data` and `iex.quote.last` *elementals*. If we now run the application when stocks are trading, then we will get something like this:

```
$ ./fizz.x64 iex.core.fizz iex.vars.fizz iex.data.fizz
fizz 0.3.0-X (20180519.2228) [x64|8|w|l]

load : loading iex.core.fizz ...
load : loading iex.vars.fizz ...
load : loaded  iex.vars.fizz in 0.003s
load : loading iex.data.fizz ...
load : loaded  iex.data.fizz in 0.005s
load : loaded  iex.core.fizz in 0.012s
load : loading completed in 0.012s
assert: iex.quote.data(AAPL, 1525976891.857000, 189.980000, 2.620000, 0.013980)
assert: iex.quote.data(GOOGL, 1525976729.736000, 1101.600000, 12.650000, 0.011620)
assert: iex.quote.data(MSFT, 1525976897.159000, 97.290000, 0.350000, 0.003610)
assert: iex.quote.data(AAPL, 1525976901.440000, 189.970000, 2.610000, 0.013930)
assert: iex.quote.data(MSFT, 1525976901.057000, 97.300000, 0.360000, 0.003710)
assert: iex.quote.data(AAPL, 1525976909.107000, 189.970000, 2.610000, 0.013930)
assert: iex.quote.data(AAPL, 1525976913.318000, 189.970000, 2.610000, 0.013930)
assert: iex.quote.data(MSFT, 1525976912.991000, 97.290000, 0.350000, 0.003610)
assert: iex.quote.data(AAPL, 1525976931.462000, 190, 2.640000, 0.014090)
assert: iex.quote.data(AAPL, 1525976934.889000, 189.990000, 2.630000, 0.014040)
```

Last known ticker's price and reactive behaviors

An easy *query* to implement and likely a common request, would be to get the latest stock price for a given stock ticker. To implement it, we would have to first retrieve the last timestamp for the stock ticker, and then the corresponding price value. We implement this in the following *procedural knowledge* `iex.quote.price`:

```
iex.quote.price {

    (:ticket,:value) :- #iex.quote.last(:ticket,:stamp), #iex.quote.data(:ticket,:stamp,:value,_,_);

}
```

As we only care about the price of the quote at that time, we use once again a *wildcard variable* for the last two *terms* of the *predicate*. The following example shows how the returned value change over consecutive calls as the data stored in the *substrate* changes in the background:

```
$ ./fizz.x64 iex.core.fizz iex.vars.fizz iex.data.fizz
fizz 0.3.0-X (20180519.2228) [x64|8|w|l]

load : loading iex.core.fizz ...
load : loading iex.vars.fizz ...
load : loaded  iex.vars.fizz in 0.003s
load : loading iex.data.fizz ...
load : loaded  iex.data.fizz in 0.005s
load : loaded  iex.core.fizz in 0.012s
load : loading completed in 0.012s
assert: iex.quote.data(AAPL, 1525977472.854000, 189.700000, 2.340000, 0.012490)
assert: iex.quote.data(GOOGL, 1525977463.290000, 1099.900000, 10.950000, 0.010060)
assert: iex.quote.data(MSFT, 1525977464.709000, 97.200000, 0.260000, 0.002680)
?- #iex.quote.price(AAPL,:v)
assert: iex.quote.data(AAPL, 1525977479.127000, 189.720000, 2.360000, 0.012600)
assert: iex.quote.data(AAPL, 1525977482.697000, 189.730000, 2.370000, 0.012650)
assert: iex.quote.data(MSFT, 1525977493.860000, 97.200000, 0.260000, 0.002680)
assert: iex.quote.data(AAPL, 1525977497.124000, 189.710000, 2.350000, 0.012540)
```

```
-> ( 189.710000 ) := 1.00 (0.003) 1
assert: iex.quote.data(AAPL, 1525977502.502000, 189.740000, 2.380000, 0.012700)
assert: iex.quote.data(MSFT, 1525977502.783000, 97.210000, 0.270000, 0.002790)
?- #iex.quote.price(AAPL,:v)
-> ( 189.740000 ) := 1.00 (0.003) 1
```

Next, let's look at how we would setup an *elemental* to watch over any changes in the price of the stock tickers, and notify us when, for instance, the value goes above or below a certain threshold value. First we are going to add in `iex.vars.fizz` a new *functional knowledge* definition which will be storing the threshold value for each of the stock tickers:

```
iex.thresholds {

    (AAPL,190);
    (GOOGL,1100);
    (MSFT,100);

}
```

We then create a new *procedural knowledge* definition in `iex.core.fizz` called `iex.quote.watch`, where each defined *prototype* will be responsible for watching over a specific situation. In this example, we will have a single one:

```
1  iex.quote.watch {
2
3      signs = {}
4
5  } {
6
7      ()  :-  @iex.quote.data(:ticker,_,:value,_,_),
8              #iex.thresholds(:ticker,:t),
9              cmp(:value,:t,:d),
10             peek(signs,:s), frm.fetch(:s,:ticker,_?[neq(:d)],0),
11             frm.store(:s,:ticker,:d,:s2), poke(signs,:s2),
12             #iex.quote.watch.report(:ticker,:value,:d),
13             hush;
14
15 }
```

Its logic is fairly simple: when a new `iex.quote.data` *statement* is asserted (*line 7*), we retrieve the corresponding threshold value (*line 8*) then compare both values (using the *primitive* `cmp`), so that the variable `d` will be bound to the value `0`, `1` or `-1`). In order to be able to only report once when the threshold is passed, we need to be able to recall the ticker's position in regard to the threshold. In this example, we are using an *elemental's property* as *memory* to store the sign of the difference for each stock tickers. We will store them in a *frame* under the key `signs`. In *line 10*, we fetch the *frame* from the *properties* with the *primitive* `peek`, then check that the value stored for the stock ticker is different from the one we are currently dealing with. If this is the first time that we are getting a quote for this stock ticker, then the *primitive* `frm.fetch` will unify it's third *term* with its fourth (which is considered to be the default value). We use as third *term* a *constrained wildcard variable* to insure that we only continue the inferencing if the sign of the difference is really different from the last one. Upon continuation, *line 11*, we store the new sign for the stock ticker in a new *frame* (along whatever else is stored in the *frame* we have read from the *properties* earlier), then use the `poke` primitive to store it in the *property*. We also use another *elemental* `iex.quote.watch.report` to compose an appropriate message to the user. This *elemental* will select the correct *prototype* to be executed according to the third *term* of the *queries* it will get (the price comparisons):

```
iex.quote.watch.report {

    (:ticker,:value,0)  :- console.puts("iex.quote.watch: ",:ticker," price at threshold");
    (:ticker,:value,1)  :- console.puts("iex.quote.watch: ",:ticker," price above threshold (",:value,")");
    (:ticker,:value,-1) :- console.puts("iex.quote.watch: ",:ticker," price below threshold (",:value,")");
}
```

Automatically saving the data

As the application runs, we may want to keep a history of the stock ticker's prices across separate executions. To support this, we are going to add an automatic saving of the `iex.quote.last` and `iex.quote.data` *knowledge* to a file. Ideally, as the number of *statements* may get large over time, it may be more suitable to deploy a more complex saving strategy (e.g. using a different set of *elementals* each day), but this isn't the subject of this article. Note, also, that saving the data to a *fizz* 's source file isn't the only option. More on this in the *user manual*.

Since it will be costly to save each time we add a new *statement*, we are going to use a second *ticker elemental* with, say, an interval value of thirty seconds. Here's the definition we are adding to `iex.core.fizz`:

```
auto.save.tick {
    class   = FZZCTicker,
    tick    = 30
} {}
```

We then need to add a second *elemental* which will use `auto.save.tick` as a *trigger*. When this occurs, it will use the `console.exec` *primitive* to request the *elementals* storing the data be saved in `iex.data.fizz`, overwriting any older data:

```
auto.save {

    () :-   @auto.save.tick(:n,_),
            console.exec(save("iex.data2.fizz",iex.quote.data,iex.quote.last)),
            hush;

}
```

Let's give this a try:

```
$ ./fizz.x64 iex.core.fizz iex.vars.fizz iex.data.fizz
fizz 0.3.0-X (20180519.2228) [x64|8|w|l]

load : loading iex.core.fizz ...
load : loading iex.vars.fizz ...
load : loading iex.data.fizz ...
load : loaded  iex.data.fizz in 0.002s
load : loaded  iex.vars.fizz in 0.004s
load : loaded  iex.core.fizz in 0.016s
load : loading completed in 0.016s
assert: iex.quote.data(AAPL, 1526068800.403000, 188.590000, -0.720100, -0.003800)
assert: iex.quote.data(GOOGL, 1526068800.466000, 1103.380000, -2.090000, -0.001890)
assert: iex.quote.data(MSFT, 1526068800.325000, 97.700000, -0.210000, -0.002140)
iex.quote.watch: AAPL price below threshold (188.590000)
iex.quote.watch: GOOGL price above threshold (1103.380000)
iex.quote.watch: MSFT price below threshold (97.700000)
save : completed in 0.001s.
```

If you let this application run for an extended period of time, it will eventually reaches a time when the stock market is closed and thus we will be saving the data unnecessarily as it would have not changed. We can fix this by using the assertions of new `iex.quote.last` *statements* as an indication that we should save the data when the tick happens. This can be easily done by modifying `auto.save` to use a *property* as a flag indicating if we should save (as new *statements* were asserted since the last time the data were saved) or skip when a tick happens:

```
auto.save {

    replies.are.triggers = no,
    save = no

```

```
6 } {
7
8     () :-   @auto.save.tick(:n,_),
9             peek(save,yes),
10            console.exec(save("iex.data.fizz",iex.quote.data,iex.quote.last)),
11            poke(save,no),
12            hush;
13
14    () :-   @iex.quote.last(_,_),
15            poke(save,yes),
16            hush;
17
18 }
```

In *line 14 to 16* we added a new *prototype* to handle the triggers from `iex.quote.last` and toggle the `save` property to `yes`. We also modified the original *prototype* to check that the same *property* has a value of `yes` when we get triggered by `auto.save.tick`, then set the value of it back to `no` once we have requested the data to be saved. If the `peek` *predicate* fails to be satisfied, the inferencing will bail and the saving will not be performed. On *line 3*, we have defined a *property* `replies.are.triggers` with a value of `no`. This instructs the *elemental* to not consider *replies* to *queries* made on the *substrate* as a *trigger*, which is the default behavior. Without this, every *query* on `iex.quote.last` will have triggered the *elemental* causing unnecessary saving to occur.

Changing the stock tickers list

Let's say that you wanted to add NVIDIA (ticker: NVDA) to the list of stocks you care about. We could just edit the source file (`iex.vars.fizz`) and modify the definition of `iex.symbols`, but it will be more practical to have a way to manage that list while the *system* is running. We can do this by introducing two new *elementals* in `iex.core.fizz`, one to be used to add a new ticker and one to remove it:

```
iex.symbols.add {

    (:s) :- #iex.symbols(:l), lst.except(:s,:l), change([iex.symbols(:l)],[iex.symbols([:s|:l])]);

}

iex.symbols.del {

    (:s) :- #iex.symbols(:l), lst.remove(:s,:l,:l2), change([iex.symbols(:l)],[iex.symbols(:l2)]);

}
```

They both work in similar way: for adding a stock ticker, the current *list* is retrieved from `iex.symbols`, then we either insure that the symbol isn't already present (with `lst.except`), and then use `change` to replace the *statement* holding the *list* with one where the new stock ticker has been appended. In the case of removing a stock ticker, we attempt the removal of it from the list (with `lst.remove`) and only if that succeeds do we replace the *statement* holding the *list* with one where the old stock ticker has been removed.

Let's give that a try:

```
$ ./fizz.x64 iex.core.fizz iex.vars.fizz iex.data.fizz
fizz 0.3.0-X (20180519.2228) [x64|8|w|l]

load : loading iex.core.fizz ...
load : loading iex.vars.fizz ...
load : loaded  iex.vars.fizz in 0.006s
load : loading iex.data.fizz ...
load : loaded  iex.data.fizz in 0.008s
load : loaded  iex.core.fizz in 0.023s
load : loading completed in 0.024s
assert: iex.quote.data(AAPL, 1526068800.403000, 188.590000, -0.720100, -0.003800)
assert: iex.quote.data(GOOGL, 1526068800.466000, 1103.380000, -2.090000, -0.001890)
assert: iex.quote.data(MSFT, 1526068800.325000, 97.700000, -0.210000, -0.002140)
```

```
iex.quote.watch: AAPL price below threshold (188.590000)
iex.quote.watch: GOOGL price above threshold (1103.380000)
iex.quote.watch: MSFT price below threshold (97.700000)
?- #iex.symbols.add(NVDA)
save : completed in 0.001s.
-> (  ) := 1.00 (0.001) 1
assert: iex.quote.data(NVDA, 1526068800.277000, 254.530000, -5.600000, -0.021530)
```

You will note that a new *statement* for the NVDA stock wasn't asserted right away, but it's only at the next periodic fetch that we get the latest known value for the new ticker. We can easily modify the application to immediately fetch the latest from the *web API* by turning the `iex.symbols` *predicate* in `iex.get` into a *trigger predicate*. We will also add the `replies.are.triggers` property to it as we do not want the *query* that are done by `iex.symbols.add` to cause the *prototype* to execute:

```
iex.query {
    replies.are.triggers = no
} {

    () :-   @iex.tick(_,_),
            @iex.symbols(:s),
            #iex.filters(:f),
            #iex.get({types=quote,symbols=:s,filter=:f},[:t,200,_,:c]),
            frm.fetch(:c,:l,:d),
            #iex.proc(:t,:l,:d),
            hush;

}
```

If we restart the application now and use `iex.symbols.add` again, the data will be fetched right away:

```
$ ./fizz.x64 iex.core.fizz iex.vars.fizz iex.data.fizz
fizz 0.3.0-X (20180519.2228) [x64|8|w|l]

load : loading iex.core.fizz ...
load : loading iex.data.fizz ...
load : loading iex.vars.fizz ...
load : loaded  iex.data.fizz in 0.003s
load : loaded  iex.vars.fizz in 0.004s
load : loaded  iex.core.fizz in 0.019s
load : loading completed in 0.020s
assert: iex.quote.data(AAPL, 1526068800.403000, 188.590000, -0.720100, -0.003800)
assert: iex.quote.data(GOOGL, 1526068800.466000, 1103.380000, -2.090000, -0.001890)
assert: iex.quote.data(MSFT, 1526068800.325000, 97.700000, -0.210000, -0.002140)
iex.quote.watch: AAPL price below threshold (188.590000)
iex.quote.watch: GOOGL price above threshold (1103.380000)
iex.quote.watch: MSFT price below threshold (97.700000)
?- #iex.symbols.add(NVDA)
-> (  ) := 1.00 (0.001) 1
assert: iex.quote.data(NVDA, 1526068800.277000, 254.530000, -5.600000, -0.021530)
```

To add a threshold value for the stock ticker we are adding, we could create a similar set of *procedural knowledge* but we could also modify `iex.symbols.add` and `iex.symbols.del` to deal with an optional threshold value:

```
iex.symbols.add {

    (:s)    :-  #iex.symbols(:l), lst.except(:s,:l),
                change([iex.symbols(:l)],[iex.symbols([:s|:l])]),
                repeal(iex.thresholds(:s,_));
    (:s,:t) :-  #iex.symbols(:l), lst.except(:s,:l),
                change([iex.symbols(:l)],[iex.symbols([:s|:l])]),
                change([iex.thresholds(:s,_)],[iex.thresholds(:s,:t)]);

}

iex.symbols.del {

    (:s)    :-  #iex.symbols(:l), lst.remove(:s,:l,:l2),
```

```
                change([iex.symbols(:l)],[iex.symbols(:l2)]),
                repeal(iex.thresholds(:s,_));

}
```

In `iex.symbols.add` we added a new *prototype* with an arity of two and add to it a call to the `change` *primitive* to replace whatever value of threshold we may have in the `iex.thresholds` *knowledge* (including none) by a new *statement* containing the new value. In `iex.symbols.del`, we just added a call to `repeal` which will remove the corresponding (if any) *statement* from `iex.thresholds`.

The last piece to add now, is the saving of the updated `iex.symbols` *knowledge*. Assuming that the list of stock tickers doesn't change often, we are going to set it up to be saved only when its content is changed. The following *knowledge* definition, which we add to `iex.core.fizz` will take care of this:

```
sync.save {
    replies.are.triggers = no
} {

    () :- @iex.symbols(:s), console.exec(save("iex.vars.fizz",iex.symbols,iex.filters,iex.thresholds));

}
```

We can then try this:

```
$ ./fizz.x64 iex.core.fizz iex.vars.fizz iex.data.fizz
fizz 0.3.0-X (20180519.2228) [x64|8|w|l]

load : loading iex.vars.fizz ...
load : loading iex.core.fizz ...
load : loaded  iex.vars.fizz in 0.002s
load : loading iex.data.fizz ...
load : loaded  iex.data.fizz in 0.002s
load : loaded  iex.core.fizz in 0.020s
load : loading completed in 0.020s
assert: iex.quote.data(AAPL, 1526068800.403000, 188.590000, -0.720100, -0.003800)
assert: iex.quote.data(GOOGL, 1526068800.466000, 1103.380000, -2.090000, -0.001890)
assert: iex.quote.data(MSFT, 1526068800.325000, 97.700000, -0.210000, -0.002140)
iex.quote.watch: AAPL price below threshold (188.590000)
iex.quote.watch: GOOGL price above threshold (1103.380000)
iex.quote.watch: MSFT price below threshold (97.700000)
?- #iex.symbols.add(NVDA,250)
-> (  ) := 1.00 (0.001) 1
save : completed in 0.008s.
assert: iex.quote.data(NVDA, 1526068800.277000, 254.530000, -5.600000, -0.021530)
iex.quote.watch: NVDA price above threshold (254.530000)
?- #iex.symbols.del(NVDA)
save : completed in 0.001s.
-> (  ) := 1.00 (0.002) 1
```

Adding some predictions

After having collected a full day worth of price variations, we now have enough data to look into turning this into something potentially insightful: We're going to build a model (using a *neural network*) which given the prices of three stocks will give us a prediction for the fourth one. Please note this is just a example of a possible use case and no warranty of any kind is made here on the real world validity of such model.

Lets assume that we have saved a full day of trading into a *fizz* source file called `iex.day.fizz`. We first need to convert it into a format that we can feed into a *network* (that is a single *statement* per training sample), which means we need to string together the price values of all four stocks at any given time. For this, let's create a new file `iex.comb.fizz` and insert the following *procedural knowledge* into it:

```
iex.comb {

    (:t0,:t1,:t2,:t3)   :-  #iex.quote.data(:t0,:time,:p0,_,_),
                            #iex.quote.data(:t1,_?[aeq(:time,20)],:p1,_,_),
                            #iex.quote.data(:t2,_?[aeq(:time,20)],:p2,_,_),
                            #iex.quote.data(:t3,_?[aeq(:time,20)],:p3,_,_),
                            bundle(iex.comb.data(:p0,:p1,:p2,:p3),1,{},4096),
                            hush;

}
```

The purpose of `iex.comb` is to combine the prices of four stocks into a single *statement*. This is done by querying `iex.quote.data` for each of the stock tickers using the timestamp we retreived from the first *predicate* as a constraint for the three others. The constraints are using the *primitive* `aeq` and will ensure that the prices we getting are not any older or more recent than 20 seconds from the timestamp we picked with the first *predicate*. When one of the tickers has a price that is too out of sync with the rest, we will drop the combination. As we have over 3000 *statements* in `iex.quote.data` and stock prices fluctuate mostly independantly, we should expect a lot of *statements* to be asserted. To minimize the cost on the *runtime*, we are using the `bundle` *primitive* to group and assert the *statements*, 4096 at a time.

Let's give this a try. Since this is going to be an intensive operation for *fizz* , we will be use the *console command* `/scan` to keep an eye on the *substrate* activity:

```
$ ./fizz.x64 iex.comb.fizz iex.day.fizz
fizz 0.3.0-X (20180519.2228) [x64|8|w|l]

load : loading iex.comb.fizz ...
load : loading iex.day.fizz ...
load : loaded  iex.comb.fizz in 0.002s
load : loaded  iex.day.fizz in 0.771s
load : loading completed in 0.772s
?- #iex.comb(NVDA,AAPL,GOOGL,MSFT)
?- /scan
scan : e:7 k:3 s:3835 p:2 u:7.88 t:2 q:2136 r:2137 z:0
scan : e:7 k:3 s:3835 p:2 u:8.13 t:2 q:2354 r:2356 z:0 (qps:872.0 rps:876.0)
scan : e:7 k:3 s:3835 p:2 u:8.38 t:3 q:2490 r:2490 z:0 (qps:541.8 rps:533.9)
scan : e:7 k:3 s:3835 p:2 u:8.63 t:1 q:2649 r:2649 z:0 (qps:641.1 rps:641.1)
scan : e:7 k:3 s:3835 p:2 u:8.88 t:5 q:2673 r:2671 z:0 (qps:94.5 rps:86.6)
scan : e:7 k:3 s:3835 p:2 u:9.13 t:1 q:2691 r:2692 z:0 (qps:73.2 rps:85.4)
scan : e:7 k:3 s:3835 p:2 u:9.38 t:4 q:2755 r:2754 z:0 (qps:252.0 rps:244.1)
...
scan : e:7 k:3 s:3835 p:2 u:22.89 t:5 q:4171 r:8022 z:0 (qps:0.0 rps:588.9)
scan : e:7 k:3 s:3835 p:2 u:23.14 t:0 q:4171 r:8235 z:0 (qps:0.0 rps:869.4)
scan : e:8 k:4 s:7931 p:2 u:23.39 t:2 q:4171 r:8383 z:0 (qps:0.0 rps:587.3)
scan : e:8 k:4 s:7931 p:2 u:23.64 t:0 q:4171 r:8578 z:0 (qps:0.0 rps:786.3)
scan : e:8 k:4 s:7931 p:2 u:23.89 t:5 q:4171 r:8755 z:0 (qps:0.0 rps:694.1)
scan : e:8 k:4 s:7931 p:2 u:24.14 t:0 q:4171 r:8915 z:0 (qps:0.0 rps:653.1)
scan : e:8 k:4 s:7931 p:2 u:24.39 t:0 q:4171 r:9003 z:0 (qps:0.0 rps:352.0)
scan : e:8 k:4 s:7931 p:2 u:24.64 t:0 q:4171 r:9003 z:0 (qps:0.0 rps:0.0)
scan : e:8 k:4 s:7931 p:2 u:24.89 t:0 q:4171 r:9003 z:0 (qps:0.0 rps:0.0)
scan : e:9 k:5 s:8669 p:2 u:25.14 t:1 q:4171 r:9003 z:0 (qps:0.0 rps:0.0)
scan : e:9 k:5 s:8669 p:2 u:25.39 t:0 q:4171 r:9003 z:0 (qps:0.0 rps:0.0)
scan : e:9 k:5 s:8669 p:2 u:25.64 t:0 q:4171 r:9003 z:0 (qps:0.0 rps:0.0)
scan : e:9 k:5 s:8669 p:2 u:25.89 t:0 q:4171 r:9003 z:0 (qps:0.0 rps:0.0)
scan : completed.
?- /stats
stats : e:9 k:5 s:8669 p:2 u:31.48 t:0 q:4171 r:9003 z:0
```

We will then save the generated data into `iex.comb.data.fizz` so that we do not have to regenerate them:

```
?- /list
list : efefdefe-7649-e84b-678b-2ef46f209d8b MRKCBFSolver        iex.comb
list : ccbe9bfd-9800-714a-3c89-b485b676e3b3 MRKCLettered        iex.comb.data
list : 61136792-7d9f-9645-6cb8-fedbc125b3b7 MRKCLettered        iex.comb.data
list : 747c1a40-d6b4-7b46-c4a5-d09fd36500e6 MRKCLettered        iex.quote.data
list : 4 elementals listed in 0.000s
?- /save("iex.comb.data.fizz",iex.comb.data)
save : completed in 0.078s.
```

Next, we are going to setup a neural network *elemental* (of class `FZZCFFBNetwork`) so that can have it learn from the practice data we just generated, and hopefully be later able to predict the price of a stock based on any three others. Create a new *fizz* source file called `iex.ffbn.fizz` and place the following definition in it:

```
iex.ffbn {
        class = FZZCFFBNetwork,
        alias = iex.ffbn,
        query = iex.comb.data(_,_,_,_),
        generalize = [[i,i,i,o],[i,i,o,i],[i,o,i,i],[o,i,i,i]],
        formatting = [d,d,d,d],
        hidden_layers = 4,
        neurons_in_hidden_layers = 12
} {}
```

The *elemental*'s properties instructs it to create four neural networks based on the *statements* it will be receiving (the `generalize` property provides a list of the mapping of the *statements*' *terms* with the inputs and outputs expected by the networks) as answers to the query it will be asking to the *substrate* (the `query` property). We also indicate (with the `formatting` property) that all *terms* are numbers (`d` stand for decimal). Finally, we indicate that each of the four networks must have 4 hidden layers, each composed of 12 neurons.

Defined as such, a `FZZCFFBNetwork` *elemental* is ready for the training samples to be provided and for the training to be executed. We do not need to process the `iex.comb.data` further as they will be automatically normalized before the training. All that is left for us, is to start *fizz* by loading `iex.comb.data.fizz` and `iex.ffbn.fizz`:

```
$ ./fizz.x64 iex.ffbn.fizz iex.comb.data.fizz
fizz 0.3.0-X (20180519.2228) [x64|8|w|l]

load : loading iex.ffbn.fizz ...
load : loading iex.comb.data.fizz ...
load : loaded  iex.ffbn.fizz in 0.003s
load : loaded  iex.comb.data.fizz in 1.371s
load : loading completed in 1.372s
?- /list
list : 877ffaaa-da0b-b749-b0b5-d749600d7e22 MRKCLettered        iex.comb.data
list : a2ce6e49-3f3a-694b-38be-cc20bb37525f MRKCLettered        iex.comb.data
list : b9d0b414-ba9a-874d-4bae-0b6e491a5ca5 FZZCFFBNetwork      iex.ffbn (iex.ffbn)
list : 3 elementals listed in 0.000s
```

To communicate with the `iex.ffbn` *elemental* outside of the common query/reply pattern, we need to use the *console command* `/tell`. The first of such commands we are going to initiate is to get the *elemental* to post the *query* and collect all answers it will receive:

```
?- /tells(iex.ffbn,acquires)
iex.ffbn - requesting training data ...
iex.ffbn : received 3778 statements
```

Once `iex.ffbn` tells us that it has received all the expected *statements* (3778 in this case), we are ready to start the training. Note that this part may take a while as the *elemental* does it on the CPU and not the GPU (as of this writing). It does, however, get executed in background threads so the *substrate* should stay responsive:

```
?- /tells(iex.ffbn,practice(0.8,1024,0.1))
iex.ffbn - training set has 3778 samples
iex.ffbn - training in progress
iex.ffbn - training in progress
iex.ffbn - training in progress
iex.ffbn - training in progress
iex.ffbn - practice completed (0.000075,0.005914) in 38.96s
iex.ffbn - practice completed (0.000222,0.007412) in 39.50s
```

```
iex.ffbn - practice completed (0.000077,0.005348) in 73.31s
iex.ffbn - practice completed (0.000112,0.005373) in 74.19s
```

The *terms* used in the `practice` *functor* specifies the training parameters: the first indicates the ratio between *training data* and *validation data* (`0.8` here means 80 percent of the 3778 *statements* will be used for training and 20 percent for *validation*). The second term is the number of *epochs* to train the model for, and finally the learning rate. For each of the trained networks, the *elemental* will output the *practice* and *validation* errors as seen above. From there, the four networks are ready to predict using the same type of query that you will use with any other *elemental*. For example:

```
?- #iex.ffbn(:p, 188.590000, 1103.380000, 97.7)
-> ( 254.693631 ) := 0.99 (0.001) 1
?- #iex.ffbn(254.640000, 188.030000, :p, 97.865000)
-> ( 1106.294365 ) := 1.00 (0.001) 1
```

To avoid having to retrain the networks, we will save them as part of the *elemental*'s properties with the *console command* `/save`:

```
?- /save("iex.ffbn.fizz",iex.ffbn)
save : completed in 0.009s.
```

If you have the curiosity to open `iex.ffbn.fizz` now, you will see that the model was encoded within a *binary term* under the *property* `data`. From then on, unless we want to retrain our model, we just need to load `iex.ffbn.fizz` to have the ability to predict:

```
$ ./fizz.x64 iex.ffbn.fizz
fizz 0.3.0-X (20180519.2228) [x64|8|w|l]

load : loading iex.ffbn.fizz ...
load : loaded  iex.ffbn.fizz in 0.014s
load : loading completed in 0.014s
?- #iex.ffbn(254.640000, 188.030000, :p, 97.865000)
-> ( 1106.294365 ) := 1.00 (0.001) 1
?- #iex.ffbn(:p, 188.590000, 1103.380000, 97.7)
-> ( 254.693631 ) := 0.99 (0.001) 1
```

To conclude this section, and this article, lets now look at integrating the model we have just build to test the validity of our predictions as we get new stock ticker prices throughout the day. To that effect, create a new *fizz* source file called `iex.predict.fizz`. First, we will write an *procedural knowledge* which, not unlike what we have done with `iex.quote.watch`, will rely on `iex.quote.data` as a trigger. For each new quote *statement* we will fetch the closest (timestamp wise) known price for the three other stock tickers and then use `iex.ffbn` to predict the price. Finally, we will report on the difference between our model and reality:

```
iex.predict {

    () :-   @iex.quote.data(NVDA,:t,:n,_,_),
            #iex.quote.data(AAPL,_?[aeq(:t,20)],:a,_,_),
            #iex.quote.data(GOOGL,_?[aeq(:t,20)],:g,_,_),
            #iex.quote.data(MSFT,_?[aeq(:t,20)],:m,_,_),
            #iex.ffbn(:x,:a,:g,:m),
            #iex.predict.report(NVDA,:t,:x,:n),
            hush;

    () :-   @iex.quote.data(AAPL,:t,:a,_,_),
            #iex.quote.data(NVDA,_?[aeq(:t,20)],:n,_,_),
            #iex.quote.data(GOOGL,_?[aeq(:t,20)],:g,_,_),
            #iex.quote.data(MSFT,_?[aeq(:t,20)],:m,_,_),
            #iex.ffbn(:n,:x,:g,:m),
            #iex.predict.report(AAPL,:t,:x,:a),
            hush;

    () :-   @iex.quote.data(GOOGL,:t,:g,_,_),
```

```
            #iex.quote.data(NVDA,_?[aeq(:t,20)],:n,_,_),
            #iex.quote.data(AAPL,_?[aeq(:t,20)],:a,_,_),
            #iex.quote.data(MSFT,_?[aeq(:t,20)],:m,_,_),
            #iex.ffbn(:n,:a,:x,:m),
            #iex.predict.report(GOOGL,:t,:x,:g),
            hush;

    () :-   @iex.quote.data(MSFT,:t,:m,_,_),
            #iex.quote.data(NVDA,_?[aeq(:t,20)],:n,_,_),
            #iex.quote.data(AAPL,_?[aeq(:t,20)],:a,_,_),
            #iex.quote.data(GOOGL,_?[aeq(:t,20)],:g,_,_),
            #iex.ffbn(:n,:a,:g,:x),
            #iex.predict.report(MSFT,:t,:x,:m),
            hush;

}
```

The *procedural knowledge* `iex.predict.report` simply computes an error value between the predicted and actual value and asserts a new *statement* so that we can later review it:

```
iex.predict.report {

    (:s,:t,:p,:a) :-    sub(:p,:a,:d),mao.abs(:d,:e),div(:e,:a,:e2),mul(:e2,100,:err),
                        #assert(iex.predict.data(:s,:t,:p,:a,:err));

}
```

With a bit more work, we can repurpose part of `iex.predict` to build a *procedural knowledge* which given a timestamp and a ticker symbol will give us the price at that time as well as the price we would have predicted based on the three others prices around the same time. Add the following definition to the same file:

```
iex.predict.price {

    (NVDA,:t,:v,:p,:e)  :-  #iex.quote.data(NVDA,:rt?[aeq(:t,30)],:v,_,_),
                            #iex.quote.data(AAPL,_?[aeq(:rt,30)],:a,_,_),
                            #iex.quote.data(GOOGL,_?[aeq(:rt,30)],:g,_,_),
                            #iex.quote.data(MSFT,_?[aeq(:rt,30)],:m,_,_),
                            #iex.ffbn(:p,:a,:g,:m),
                            sub(:p,:v,:d), mao.abs(:d,:abs), div(:abs,:v,:e2), mul(:e2,100,:e);

    (AAPL,:t,:v,:p,:e)  :-  #iex.quote.data(AAPL,:rt?[aeq(:t,30)],:v,_,_),
                            #iex.quote.data(NVDA,_?[aeq(:rt,30)],:n,_,_),
                            #iex.quote.data(GOOGL,_?[aeq(:rt,30)],:g,_,_),
                            #iex.quote.data(MSFT,_?[aeq(:rt,30)],:m,_,_),
                            #iex.ffbn(:n,:p,:g,:m),
                            sub(:p,:v,:d), mao.abs(:d,:abs), div(:abs,:v,:e2), mul(:e2,100,:e);

    (GOOGL,:t,:v,:p,:e) :-  #iex.quote.data(GOOGL,:rt?[aeq(:t,30)],:v,_,_),
                            #iex.quote.data(NVDA,_?[aeq(:rt,20)],:n,_,_),
                            #iex.quote.data(AAPL,_?[aeq(:rt,20)],:a,_,_),
                            #iex.quote.data(MSFT,_?[aeq(:rt,20)],:m,_,_),
                            #iex.ffbn(:n,:a,:p,:m),
                            sub(:p,:v,:d), mao.abs(:d,:abs), div(:abs,:v,:e2), mul(:e2,100,:e);


    (MSFT,:t,:v,:p,:e) :-   #iex.quote.data(MSFT,:rt?[aeq(:t,30)],:v,_,_),
                            #iex.quote.data(NVDA,_?[aeq(:rt,20)],:n,_,_),
                            #iex.quote.data(AAPL,_?[aeq(:rt,20)],:a,_,_),
                            #iex.quote.data(GOOGL,_?[aeq(:rt,20)],:g,_,_),
                            #iex.ffbn(:n,:a,:g,:p),
                            sub(:p,:v,:d), mao.abs(:d,:abs), div(:abs,:v,:e2), mul(:e2,100,:e);

}
```

Here's an example using stock prices collected on May 16th:

```
$ ./fizz.x64 iex.data.051618.fizz iex.predict.fizz iex.ffbn.fizz
fizz 0.3.0-X (20180519.2228) [x64|8|w|l]
```

```
load : loading iex.data.051618.fizz ...
load : loading iex.predict.fizz ...
load : loaded  iex.predict.fizz in 0.017s
load : loading iex.ffbn.fizz ...
load : loaded  iex.ffbn.fizz in 0.010s
load : loaded  iex.data.051618.fizz in 1.135s
load : loading completed in 1.136s
?- #iex.predict.price(MSFT,1526484458,:v,:p,:e)
-> ( 96.720000 , 97.849645 , 1.167954 ) := 0.99 (0.057) 1
-> ( 96.720000 , 97.848788 , 1.167067 ) := 0.99 (0.062) 2
-> ( 96.720000 , 97.852346 , 1.170747 ) := 0.99 (0.064) 3
-> ( 96.720000 , 97.856259 , 1.174792 ) := 0.99 (0.067) 4
-> ( 96.720000 , 97.857295 , 1.175864 ) := 0.99 (0.072) 5
-> ( 96.720000 , 97.849304 , 1.167602 ) := 0.99 (0.082) 6
-> ( 96.720000 , 97.848452 , 1.166721 ) := 0.99 (0.087) 7
-> ( 96.720000 , 97.851987 , 1.170375 ) := 0.99 (0.089) 8
-> ( 96.720000 , 97.855874 , 1.174394 ) := 0.99 (0.095) 9
-> ( 96.720000 , 97.856904 , 1.175459 ) := 0.99 (0.097) 10
?- #iex.predict.price(AAPL,1526484458,:v,:p,:e)
-> ( 187.180000 , 188.071139 , 0.476087 ) := 1.00 (0.052) 1
-> ( 187.180000 , 188.071807 , 0.476444 ) := 1.00 (0.052) 2
-> ( 187.180000 , 188.071236 , 0.476139 ) := 1.00 (0.067) 3
-> ( 187.180000 , 188.071905 , 0.476496 ) := 1.00 (0.067) 4
-> ( 187.180000 , 188.071406 , 0.476229 ) := 1.00 (0.073) 5
-> ( 187.180000 , 188.072076 , 0.476587 ) := 1.00 (0.073) 6
-> ( 187.180000 , 188.071382 , 0.476216 ) := 1.00 (0.079) 7
-> ( 187.180000 , 188.072052 , 0.476574 ) := 1.00 (0.079) 8
```