# Building a simple room exploring autonomous robot with *fizz* and *LEGO® Mindstorms® EV3*

Jean-Louis Villecroze
jlv@f1zz.org @CocoaGeek

June 21, 2019

**Abstract**

In this article[1], we will detail the implementation of a simple room exploring autonomous robot, built from *LEGO® Mindstorms®* and running on the *EV3 Intelligent brick*.

## Prerequisite

A basic understanding of the concepts behind *fizz* (version 0.6 and up) is expected from the reader of this article. It is suggested to read the introductory article *Building a simple stock prices monitor with fizz* [2] first or at least read sections two to four of the *user manual* for an overview of the language and runtime. The complete source code discussed in this article can be downloaded from the author's web site [3].

## Running *fizz* on the *LEGO® Mindstorms®*

Let's get started by seeing how to run *fizz* on the *EV3 Intelligent brick*. First, you need an SD card to flash the custom Linux distribution called *ev3dev*[4]. Once you have flashed the image and inserted the card in the SD port of the brick, plug in a USB WiFi dongle in the unit and press the middle button to get it booting. If the SD card is bootable, the *EV3* will boot from it instead of booting the standard Mindstorms OS from its internal flash storage. The *EV3*, being far from a workhorse, will take some time to boot and display the *Brickman* UI which you will need to use to setup the WiFi connection. The setup will be saved so you won't have to do that again. Once the unit is connected to your local network, you can use `ssh` to log into it (the default user for *ev3dev* is `robot` with the password is `maker`) and install the *Linux* build of *fizz* :

```
jlv@arrakis:~  ssh robot@192.168.1.21
Password:
Linux ev3dev 4.14.96-ev3dev-2.3.2-ev3 #1 PREEMPT Sun Jan 27 21:27:35 CST 2019 armv5tejl
             _____     _
   _____   _|___ /  __| | _____   __
  / _ \ \ / / |_ \ / _` |/ _ \ \ / /
 |  __/\ V / ___) | (_| |  __/\ V /
  \___| \_/ |____/ \__,_|\___| \_/

Debian stretch on LEGO MINDSTORMS EV3!
Last login: Fri Jun  7 02:09:10 2019 from 192.168.1.29
robot@ev3dev:~  wget http://f1zz.org/downloads/fizz.0.6.0-X-LNX.tgz
--2019-06-07 03:08:29--  http://f1zz.org/downloads/fizz.0.6.0-X-LNX.tgz
Resolving f1zz.org (f1zz.org)... 149.56.222.2
Connecting to f1zz.org (f1zz.org)|149.56.222.2|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 24741747 (24M) [application/x-tar]
Saving to: fizz.0.6.0-X-LNX.tgz

fizz.0.6.0-X-LNX.tgz 100%[=========================================================================>]  23.59M   724KB/s
     in 34s

2019-06-07 03:09:04 (701 KB/s) - fizz.0.6.0-X-LNX.tgz saved [24741747/24741747]

robot@ev3dev:~  tar xvzf fizz.0.6.0-X-LNX.tgz
robot@ev3dev:~  cd fizz.0.6.0-X
robot@ev3dev:~/fizz.0.6.0-X  ./fizz.ev3
fizz 0.6.0-X (20190601.1943) [lnx.ev3|1]
Press the ESC key at anytime for input prompt
```

[1] Thanks to Robert Wasmann (@retrospasm) for providing feedback and reviewing this document.
[2] http://f1zz.org/downloads/iex.pdf
[3] http://f1zz.org/downloads/ev3.tgz
[4] https://www.ev3dev.org/

As the *EV3*'s hardware is more limited (`arm926ej-s`) than more recent embedded boards, *fizz* has a special build for that platform (`fizz.ev3`) where not all modules are available (notably `LGR` and `WWW`). Keep in mind also that performance is also lacking.

We can test that *fizz* is running with one of the simpler samples:

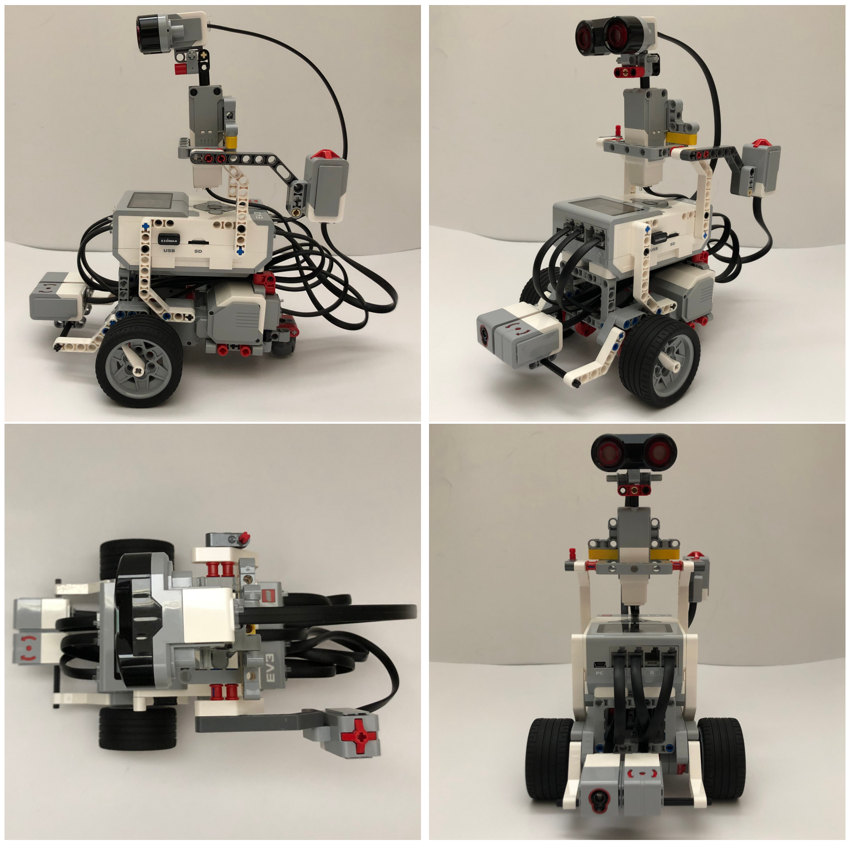
```
robot@ev3dev:~/fizz.0.6.0-X  ./fizz.ev3 ./etc/samples/calc.fizz
fizz 0.6.0-X (20190601.1943) [lnx.ev3|1]
Press the ESC key at anytime for input prompt

load : loading ./etc/samples/calc.fizz ...
load : loaded ./etc/samples/calc.fizz in 1.236s
load : loading completed in 1.367s
?- #calc([[5,mul,2],mul,[1,add,:v]],130)
-> ( 12 ) := 1.00 (0.903) 1
```

Just running *fizz* on the *EV3* isn't enough to have access to the unit's sensors and motors. A `EV3` module needs to be loaded in *fizz* and some specific *elementals* must be running on the *substrate* for *fizz* to be accessing the unit capabilities.

## The robot

We are going to use a pretty standard design for a mobile robot which will only use the blocks that comes with the *Education* version of the *LEGO® Mindstorms®* kit[5]. In this article, we won't be describing the actual step-by-step assembly of the robot, but here are a couple of pictures to give you an idea of how it was put together:



[5]https://education.lego.com/en-us/middle-school/intro/mindstorms-ev3

Here is the list of the sensors and motors you will need to recreate it:

| | |
|---|---|
| 2 | LEGO EV3 Large Servo Motor |
| 1 | LEGO EV3 Medium Servo Motor |
| 1 | LEGO EV3 Ultrasonic Sensor |
| 1 | LEGO EV3 Gyro Sensor |
| 1 | LEGO EV3 Color Sensor |
| 1 | LEGO EV3 Touch Sensor |

With this in mind, the basic functions the robot has are as follows:

- use the two medium motors for *tank steering* type mobility
- use the gyroscope to keep track of the heading
- use an orientable sonic sensor to look for obstacles
- use a color sensor to sense close proximity of a low obstacle (since the Sonic sensor is higher than the main body of the robot)
- use the touch sensor as a (software) power button

Altough, the software should run entirely on the *EV3 Intelligent brick*, we will make provisions for it to use the clusterisation capability of *fizz* so that parts of the logic can be run outside of the physical robot on a much faster, and separate computer.

Lastly, as it is painfully slow to use `vi` directly on the *EV3*, what follows assumes that you will be using a *MacOS* or *Linux* computer with your favorite code editor and copying the files over to the *EV3* placing them in `/home/robot/fizz.0.6.0-X/etc/ev3`.

## ***Predicates*** pattern

As all of the *elementals* provided by the `EV3` module follows the same pattern when it comes to interacting with them, we are going to follow that pattern and apply it to all the *elementals* we may be creating. That is, we will use specific *predicates* to read or write values (via `peek`, `poke`) and execute (or cancel) specific *functions* (via `call`, `halt`).

Each *predicate* will have two *terms*: a *symbol* (either `peek`, `poke`, `call` or `halt`) followed by a *functor* or a *list* of *functors*. For examples:

| | |
|---|---|
| `#ev3.something(poke,[value.a(30),value.b(hello)])` | set the values of `value.a` and `value.b`. |
| `#ev3.something(peek,value.a(:a))` | get the value of `value.a`. |
| `#ev3.something(call,do.it(45))` | start the `do.it` function with `45` as argument. |
| `#ev3.something(halt)` | abort any running function. |

Similarly, when an *elemental* will be publishing something, it will use the same pattern with the *symbol* `hint` as the first *term* in the *statement*. For example:

| | |
|---|---|
| `ev3.sen.touch(hint, pressed(1))` | the *Touch Sensor* is pressed down. |
| `ev3.sen.touch(hint, pressed(0))` | the *Touch Sensor* was released. |

## System, motors and sensors

To get started, let's create a first *fizz* file that will describe the core *elementals* that are necessary for the `EV3` module to be usable. We will call this file `system.fizz`:

```
 1 ev3.sys {
 2 
 3     class = EV3CSYSLEGOSystem
 4 
 5 }
 6 
 7 ev3.sys.led.0 {
 8 
 9     class = EV3CSYSLEGOLed,
10     index = 0
11 
12 }
13 
14 ev3.sys.led.1 {
15 
16     class = EV3CSYSLEGOLed,
17     index = 1
18 
19 }
```

In it, we define three *elementals* each mapped to a specific class of *elemental* that is provided by the `EV3` module. On line 1, we define `ev3.sys` which along with providing a way to read the device's battery status, will watch over plugging and unplugging of sensors and motors. It also provides some core functionalities for the other *elementals* in the modules. On lines 7 and 14, we define the *elementals* that control the two LEDs available on the *EV3*. The `index` property indicates which of the LEDs (`0` is the left one) the *elemental* uses.

To try this, we need to create a *solution* file (JSON formatted) that can be loaded by *fizz* and load `system.fizz` as well as any modules we may be using. Create that file with the name `robot.json` and copy the following content into it. It simply indicates the modules and the source files to be loaded:

```
{
    "solution" : {
        "modules" :   ["modEV3"],
        "sources" :   ["system.fizz"],
        "globals" :   []
    }
}
```

If we now use that file with *fizz* , we can query the battery status and change the LEDs brightness. Note that each of the LEDs is actually build from two physical LEDs: one green and one red.

```
robot@ev3dev:~/fizz.0.6.0-X  ./fizz.ev3 ./etc/ev3/robot.json
fizz 0.6.0-X (20190601.1943) [lnx.ev3|1]
Press the ESC key at anytime for input prompt

load : loading ./etc/ev3/robot.json ...
load : loaded ./mod/lnx/ev3/modEV3.so in 0.125s
load : loading ./etc/ev3/system.fizz ...
load : loaded ./etc/ev3/system.fizz in 0.253s
load : loading completed in 0.515s
?- #ev3.sys(peek,bat.voltage.p(:b))
-> ( 0.440722 ) := 1.00 (0.068) 1
?- #ev3.sys.led.0(poke,g(0.5))
-> (  ) := 1.00 (0.028) 1
?- #ev3.sys.led.1(poke,r(1))
-> (  ) := 1.00 (0.025) 1
```

We are now going to describe the sensor layout for the robot by defining the required *elementals* in a new *fizz* file which we will call `sensors.fizz`. We first define the *elemental* handling the *LEGO* Touch sensor (which we will be using to power ON or OFF the robot):

```
1 ev3.sen.touch {
2 
3     class   = EV3CSENLEGOTouch,
4     port    = port2,
5     verbose = yes
6 
7 }
```

On line 4, we indicate which port of the *EV3* unit the sensor is connected to. If you have connected it to another one of the four possible ports, you will have to modify it there (possible choices are `port1`, `port2`, `port3` and `port4`). The following property we set is `verbose`. This is a common property for an *elemental* which when set to `yes` indicates that the *elemental* should output some traces during its execution so that we get a better sense of what is going on. In this case, mainly if the physical sensor is detected on `port2` or not.

We will now define the *elemental* for the *LEGO* Color sensor:

```
1 ev3.sen.color {
2 
3     class   = EV3CSENLEGOColor,
4     port    = port4,
5     mode    = reflected,
6     verbose = yes
7 
8 }
```

Just like the previous *elemental*, we specify the `port` and `verbose` property but also (on line 5) indicates the `mode` in which the sensor must operate. Here, we pick the `reflected` mode which is the most suited for close proximity detection as the amount of reflected light will increase as the robot gets closer to a reflecting surface.

Moving on to the *LEGO* Ultrasonic sensor:

```
1 ev3.sen.sonic {
2 
3     class   = EV3CSENLEGOSonic,
4     port    = port3,
5     mode    = continuous,
6     verbose = yes
7 
8 }
```

Here also, we set the `mode` in which the sensor will operate. The `continuous` mode will have the sensor continuously sensing the distance to the obstacle in its line-of-sight.

Lastely, we define the *LEGO* Gyroscope sensor:

```
1 ev3.sen.gyros {
2 
3     class   = EV3CSENLEGOGyros,
4     port    = port1,
5     mode    = angle1axis,
6     verbose = yes
7 
8 }
```

As we are using it to sense the orientation around the vertical axis of the robot, we will use the mode `angle1axis`. For more details on this sensor's (or others) modes, refer to the *fizz* manual.

Once we add the new *fizz* file to our solution (`robot.json`):

```
{
    "solution" : {
        "modules" :   ["modEV3"],
        "sources" :   ["system.fizz","sensors.fizz"],
        "globals" :   []
    }
}
```

We can relaunch *fizz* and start reading from the sensors. We'll use the *command* `spy` to see the *statements* published by `ev3.sen.touch` when the button is pressed. Note also that before the second reading from the gyroscope, the robot was rotated by 90 degrees to its left:

```
robot@ev3dev:~/fizz.0.6.0-X  ./fizz.ev3 ./etc/ev3/robot.json
fizz 0.6.0-X (20190601.1943) [lnx.ev3|1]
Press the ESC key at anytime for input prompt

load : loading ./etc/ev3/robot.json ...
load : loaded ./mod/lnx/ev3/modEV3.so in 0.082s
load : loading ./etc/ev3/system.fizz ...
load : loaded ./etc/ev3/system.fizz in 0.193s
load : loading ./etc/ev3/sensors.fizz ...
ev3.sen.touch : sensor detected!
ev3.sen.color : sensor detected!
ev3.sen.sonic : sensor detected!
load : loaded ./etc/ev3/sensors.fizz in 0.563s
load : loading completed in 1.004s
ev3.sen.gyros : sensor detected!
?- /spy(append,ev3.sen.touch)
spy : observing ev3.sen.touch
spy : S ev3.sen.touch(hint, pressed(1)) (15.000000)
spy : S ev3.sen.touch(hint, pressed(0)) (15.000000)
?- #ev3.sen.color(peek,value(:v))
-> ( 0.010000 ) := 1.00 (0.044) 1
?- #ev3.sen.sonic(peek,value(:v))
-> ( 0.321000 ) := 1.00 (0.039) 1
?- #ev3.sen.gyros(peek,value(:v))
-> ( 0 ) := 1.00 (0.044) 1
?- #ev3.sen.gyros(peek,value(:v))
-> ( -93 ) := 1.00 (0.037) 1
```

For the three motors we are using on the robot, we are going to create a new file called `motors.fizz` and define in it an *elemental* for each one:

```
ev3.act.motor.l {

    class    = EV3CACTLEGOMotor,
    port     = portA,
    speed    = 90,
    verbose  = yes

}

ev3.act.motor.r {

    class    = EV3CACTLEGOMotor,
    port     = portD,
    speed    = 90,
    verbose  = yes

}

ev3.act.motor.t {

    class       = EV3CACTLEGOMotor,
    port        = portC,
    speed       = 270,
    stopaction  = hold,
    verbose     = yes

}
```

As the `EV3` module doesn't differentiate between medium and small motors, we will use the same class of *elemental* for each of the *tacho motors*. Using the property `port` we specify where each one is plugged in. You may need to adjust that as needed for your robot (possible choices are `portA`, `portB`, `portC` and `portD`). For each of the motors, we use the `speed` property to set the default rotational speed of the motor (expressed in degree per second). The two *elementals* `ev3.act.motor.l` and `ev3.act.motor.r` are the motors that will be used to drive the robot around, while `ev3.act.motor.t` will be used to change the ultrasonic sensor orientation. For the later, we specify the property `stopaction` to define what it should do when the requested position is reached. Here, we will use the `hold` mode as we want the motor to hold its position. This is needed as there may be some resistance coming from the cable connecting the sonic sensor to the *EV3* in some orientations.

Once we add `motors.fizz` to our solution file (`robot.json`):

```
{
    "solution" : {
        "modules" :   ["modEV3"],
        "sources" :   ["system.fizz","sensors.fizz","motors.fizz"],
        "globals" :   []
    }
}
```

We can relaunch *fizz* and test sending commands to the top motor:

```
robot@ev3dev:~/fizz.0.6.0-X  ./fizz.ev3 ./etc/ev3/robot.json
fizz 0.6.0-X (20190601.1943) [lnx.ev3|1]
Press the ESC key at anytime for input prompt

load : loading ./etc/ev3/robot.json ...
load : loaded ./mod/lnx/ev3/modEV3.so in 0.082s
load : loading ./etc/ev3/system.fizz ...
load : loaded ./etc/ev3/system.fizz in 0.205s
load : loading ./etc/ev3/sensors.fizz ...
ev3.sen.touch : sensor detected!
ev3.sen.color : sensor detected!
ev3.sen.sonic : sensor detected!
load : loaded ./etc/ev3/sensors.fizz in 0.569s
load : loading ./etc/ev3/motors.fizz ...
ev3.sen.gyros : sensor detected!
ev3.act.motor.l : motor detected!
ev3.act.motor.r : motor detected!
load : loaded ./etc/ev3/motors.fizz in 0.449s
ev3.act.motor.t : motor detected!
load : loading completed in 1.574s
?- #ev3.act.motor.t(call,by(-45))
-> (  ) := 1.00 (0.036) 1
?- #ev3.act.motor.t(peek,position(:p))
-> ( -44 ) := 1.00 (0.036) 1
```

## Heartbeat and power button

Let's continue enabling our robot, by making use of the *LEGO* Touch sensor and LEDs. Whenever the robot is *powered* on (that is when it is allowed to move) we are going to make the left LED blink to indicate that the robot is up and running.

Create a new *fizz* file called `heartbeat.fizz`. In it, we will define the *elemental* responsible by turning the left LED ON and OFF periodically:

```
ev3.bev.hbeat {

    chatty                 = no,
    replies.are.triggers   = no,
    power                  = 0,
    toggle                 = 1
```

```
 7
 8 } {
 9
10     ()  :-  @ev3.tck.fast(_,_),
11             peek(power,1),
12             boo.not( toggle,:toggle),
13             poke(toggle,:toggle),
14             #ev3.sys.led.0(poke,g(:toggle)),
15             hush;
16
17
18     ()  :-  @ev3.bev.state(hint,power(off)), poke(power,0), #ev3.sys.led.0(poke,g(0));
19     ()  :-  @ev3.bev.state(hint,power(on)),  poke(power,1), #ev3.sys.led.0(poke,g(1));
20
21 }
```

The file defines three trigger based *prototypes*. The first one (in line 10) will toggle the LED ON and OFF by querying the *elemental* `ev3.sys.led.0` with the requiered brightness value. The trigger *predicate* (`ev3.tck.fast`) references an *elemental* that we will add shortly. On line 11, we look at the value of the `power` property which will be a local cache of the power state of the robot. The *prototypes* on line 18 and 19 will change the value of the property on reaction to trigger *statements* coming from the yet to be defined *elemental* `ev3.bev.state`. This *elemental* will keep track of the power state of the robot. When that is changed, for example when the user presses the Touch sensor, the *elemental* will publish a *statement* (either `ev3.bev.state(hint,power(off))` or `ev3.bev.state(hint,power(on))`) which will then trigger `ev3.bev.hbeat`. The property `power` will then get set and the LED will get turned either ON or OFF. Line 12 and 13 shows the logic used to toggle the brightness value between 0 and 1 at each periodic trigger coming from `ev3.tck.fast`.

Just in case you are wondering what is that `hush` *primitive* we called on line 15 (you will see it used later on as well), know that it is only an optional performance improvement. The *primitive* will suppress the publication of a `ev3.bev.hbeat` *statement* on successful conclusion of the inference, saving a bit of the runtime resources, which matters on a processing power limited board like the *EV3*.

Let's now define `ev3.tck.fast` in a new file called `ticks.fizz`:

```
ev3.tck.fast {
    class           = FZZCTicker,
    tick            = 0.5,
    tick.on.attach  = yes
} {}
```

It is defined as an *elemental* of class `FZZCTicker`, and will publish a *statement* (of arity two) at a given pace which we will set to 0.5 (seconds). We also indicate (with the `tick.on.attach` property) that we want the *elemental* to publish a *statement* when it is added to the *substrate* rather than wait for the first time tick.

Next, let's define the `ev3.bev.state` *elemental* in a separate file called `behaviors.fizz`:

```
ev3.bev.state {

    power = off

} {

    (peek,power(:state))^   :- peek(power,:state);
    (poke,power(:state))^   :- poke(power,:state), declare( self,[hint,power(:state)]);

    ()  :-  @ev3.sen.touch(hint,pressed(1)), peek(power,on)^, ~self(poke,power(off)), hush;
    ()  :-  @ev3.sen.touch(hint,pressed(1)), peek(power,off), ~self(poke,power(on)),  hush;

}
```

As we have seen earlier, the core purpose of this *elemental* is to keep track of when the robot is allowed to move. Changing this state can be done via a query to the *elemental* or by pressing the Touch sensor. On line 7, we define a *prototype* which when matched with a *predicate* will read the value of the `power` property. On line 8, we define the *prototype* for when the value of the property `power` is to be set. When this *prototype* gets executed, it will complete with the `declare` *primitive* which will publish a new *statement* constructed from the *terms* passed to it. `$self` is a constant which unifies to the label of the *elemental* in which it is used. This published *statement* is the one we set up a trigger *predicate* for in `ev3.bev.hbeat`. On line 10 and 11, we define the two trigger based *prototypes* which will react to a press event on the Touch sensor (when the sensor is pressed down, the *functor* that is the second *term* of the published *statement* will unify to `pressed(1)`. When depressed, it will be `pressed(0)`).

Once we add the three new files to our solution, we'll be ready to test the transition of the `power` state for the robot:

```
{
    "solution" : {
        "modules" :   ["modEV3"],
        "sources" :   ["system.fizz","sensors.fizz","motors.fizz","ticks.fizz","heartbeat.fizz","behaviors.fizz"],
        "globals" :   []
    }
}
```

As we did before, we will use the `spy` *command* to observe the state transition as well as the Touch sensor presses:

```
robot@ev3dev:~/fizz.0.6.0-X  ./fizz.ev3 ./etc/ev3/robot.json
fizz 0.6.0-X (20190601.1943) [lnx.ev3|1]
Press the ESC key at anytime for input prompt

load : loading ./etc/ev3/robot.json ...
load : loaded ./mod/lnx/ev3/modEV3.so in 0.081s
load : loading ./etc/ev3/system.fizz ...
load : loaded ./etc/ev3/system.fizz in 0.188s
load : loading ./etc/ev3/sensors.fizz ...
ev3.sen.touch : sensor detected!
ev3.sen.color : sensor detected!
ev3.sen.sonic : sensor detected!
load : loaded ./etc/ev3/sensors.fizz in 0.564s
load : loading ./etc/ev3/motors.fizz ...
ev3.sen.gyros : sensor detected!
ev3.act.motor.l : motor detected!
ev3.act.motor.r : motor detected!
load : loaded ./etc/ev3/motors.fizz in 0.424s
ev3.act.motor.t : motor detected!
load : loading ./etc/ev3/ticks.fizz ...
load : loaded ./etc/ev3/ticks.fizz in 0.097s
load : loading ./etc/ev3/heartbeat.fizz ...
load : loaded ./etc/ev3/heartbeat.fizz in 0.387s
load : loading ./etc/ev3/behaviors.fizz ...
load : loaded ./etc/ev3/behaviors.fizz in 1.115s
load : loading completed in 3.233s
?- /spy(append,ev3.sen.touch)
spy : observing ev3.sen.touch
?- /spy(append,ev3.bev.state)
spy : observing ev3.bev.state
spy : S ev3.sen.touch(hint, pressed(1)) (15.000000)
spy : S ev3.bev.state() := 0.00 (14.980050)
spy : S ev3.bev.state(hint, power(on)) (15.000000)
spy : S ev3.sen.touch(hint, pressed(0)) (15.000000)
spy : S ev3.sen.touch(hint, pressed(1)) (15.000000)
spy : S ev3.bev.state(hint, power(off)) (15.000000)
spy : S ev3.sen.touch(hint, pressed(0)) (15.000000)
```

## Setting-up a Cluster

The native way for multiple instances of *fizz* running on separate hosts to collaborate is to use the `CLU` module. In this case, we want the robot and a main computer to be connected so that we can not only expand the computing abilities of the robot by using external resources, but also to be able to observe the execution of the solution on the robot.

Hooking this up in *fizz* is fairly simple (don't let the number of properties scare you). Create a new file called `network.fizz`. We will define the `CLU` provided *elemental* that creates the bridge between our two remote instances of *fizz* :

```
ev3.sys.network {

    class       = FZZCCLUGateway,

    filters     =   [
                        ev3.sys,
                        ev3.act.motor.t, ev3.act.motor.l, ev3.act.motor.r,
                        ev3.sen.color, ev3.sen.sonic, ev3.sen.gyros,
                        ev3.bev.state
                    ],

    MCAddress   = "233.252.1.32",
    CLUDPPort   = 49152,
    TXUDPPort   = 49153,

    Bandwidth.value = 12500,
    Bandwidth.peers = 2,
    Bandwidth.limit = 95,

    CLCadence   = 350,
    CLTimeout   = 750,
    XXTimeout   = 25,
    TXTimeout   = 500,
    SyCadence   = 1000,
    TXCadence   = 3,
    PkBLength   = 1472,
    PkRetries   = 10,
    PkWinSize   = 10,
    RXCadence   = 3

}
```

The properties that will most matters to you are `filters` and `Bandwidth.value`. The rest are out of the scope of this article. For more details, check out the *fizz* user manual. Because the amount of inferences on a *substrate* can be large, the `filters` property allows to specify which can send (and receive) to/from the other *fizz* instances that are in the *cluster* (which, by the way, is identified by the multicast address provided with the property `MCAddress`). If the label of a *predicate* or *statement* isn't in this *list*, it will not be transmitted or received. As for `Bandwidth.value`, it is the bandwidth available for the cluster (in bytes per ms). Depending on your networking setup and quality (router, USB WiFi dongle ...) you may have to adjust the value if you notice long delays when doing inferences that reach out onto the cluster.

Let's give this a try now. First, we need to modify our solution to load the `CLU` module and the new file we just created:

```
{
    "solution" : {
        "modules" :   ["modEV3","modCLU"],
        "sources" :   [ "system.fizz","sensors.fizz","motors.fizz","ticks.fizz","heartbeat.fizz",
                        "behaviors.fizz","network.fizz"],
        "globals" :   []
    }
}
```

We also need to create a new solution file for the *fizz* instance we are going to run on a desktop (or laptop). Let's call this file `host.json`. All we need on the computer to do for now is to load the same module `CLU` as well as `network.fizz`:

```
{
    "solution" : {
        "modules" :   ["modCLU"],
        "sources" :   ["network.fizz"],
        "globals" :   []
    }
}
```

Let's give this a try, first by running *fizz* on the *EV3*:

```
robot@ev3dev:~/fizz.0.6.0-X  ./fizz.ev3 ./etc/ev3/robot.json
fizz 0.6.0-X (20190601.1943) [lnx.ev3|1]
Press the ESC key at anytime for input prompt

load : loading ./etc/ev3/robot.json ...
load : loaded ./mod/lnx/ev3/modEV3.so in 0.080s
load : loaded ./mod/lnx/ev3/modCLU.so in 0.031s
load : loading ./etc/ev3/system.fizz ...
load : loaded ./etc/ev3/system.fizz in 0.186s
load : loading ./etc/ev3/sensors.fizz ...
ev3.sen.touch : sensor detected!
ev3.sen.color : sensor detected!
ev3.sen.sonic : sensor detected!
load : loaded ./etc/ev3/sensors.fizz in 0.572s
load : loading ./etc/ev3/motors.fizz ...
ev3.sen.gyros : sensor detected!
ev3.act.motor.l : motor detected!
ev3.act.motor.r : motor detected!
load : loaded ./etc/ev3/motors.fizz in 0.453s
ev3.act.motor.t : motor detected!
load : loading ./etc/ev3/ticks.fizz ...
load : loaded ./etc/ev3/ticks.fizz in 0.090s
load : loading ./etc/ev3/heartbeat.fizz ...
load : loaded ./etc/ev3/heartbeat.fizz in 0.391s
load : loading ./etc/ev3/behaviors.fizz ...
load : loaded ./etc/ev3/behaviors.fizz in 1.130s
load : loading ./etc/ev3/network.fizz ...
load : loaded ./etc/ev3/network.fizz in 0.382s
load : loading completed in 3.839s
```

We will then launch the instance on the computer and query the Ultrasonic sensor from it:

```
jlv@arrakis:~/Code/okb/apps/fizz  ./fizz.x64 ./etc/ev3/host.json
fizz 0.6.0-X (20190601.2228) [lnx.x64|8|1]
Press the ESC key at anytime for input prompt

load : loading ./etc/ev3/host.json ...
load : loaded ./mod/lnx/x64/modCLU.so in 0.000s
load : loading ./etc/ev3/network.fizz ...
load : loaded ./etc/ev3/network.fizz in 0.003s
load : loading completed in 0.004s
?- #ev3.sen.sonic(peek,value(:v))
-> ( 2.550000 ) := 1.00 (0.182) 1
```

## Drive behavior

Now that we have all the basic components in place, let's add a more advanced component to our robot; one that combines sensors and motors to perform *tank steering* type mobility. While this is something that could be implemented directly in *fizz* , for performance reasons, the `EV3` module provides an *elemental* class (`EV3CBEVDrive`) for that single purpose. We are going to make use of it.

The *elemental* ties up two motors and a Gyroscope sensor to provide an easy way to ask the robot to turn in any direction and drive or alter its course while driving. It also implements a very basic odometry system which can be used to know the rough estimated position of the robot. This can be controlled at runtime via a set of specific queries.

Re-open the `behaviors.fizz` file we created earlier, and copy into it the following definition:

```
ev3.bev.drive {

    class   = EV3CBEVDrive,

    ticks   = 150,              // control loop frequency (in ms)
    hints   = 3,                // how often to publish a hint when running a program (modulo)

    gyros   = ev3.sen.gyros,    // label of the gyros sensor
    motor.l = ev3.act.motor.l,  // label of the left motor
    motor.r = ev3.act.motor.r,  // label of the right motor

    odometry    = {             // odometry characteristics
        wheel.c =  0.176,       // circumference of the wheel (in m)
        motor.d =  0.12         // measured distance in between the center of the motors (in m)
    },

    move    = {                 // 'move' program setup
        speed   = 270,          // speed to be applied to the motors at full power level
        pid.Kp  = 3.5,          // PID's proportional constant
        pid.Kd  = 0.5,          // PID's derivative constant
        pid.Ki  = 0             // PID's integral constant
    },

    turn    = {
        speed   = 235,          // speed to be applied to the motors at full power level
        pid.Kp  = 3.5,          // PID's proportional constant
        pid.Kd  = 0.5,          // PID's derivative constant
        pid.Ki  = 0             // PID's integral constant
    }

} {}
```

For more details on the working of this *elemental* and the meaning of some of its properties, check out the *fizz* user manual. I did leave comments on each line for the curious readers. For now, though, we are only going to focus on the properties that are more likely to be changed to adapt to the robot that you have assembled; that is the value specified in the *frame* assigned to the `odometry` property. Take out a ruler and mesure the circumference of the wheel attached to the two motors as well as the distance in between the center of the two motors. Convert both values in meters and update the value for `wheel.c` and `motor.d`. Both values are crucial for the odometry estimation.

If you named the gyroscope sensor or the motors *elementals* differently, you will need to reflect that change to the properties `gyros`, `motor.l` and `motor.r`.

Here are examples of the *predicates* to which the *elemental* will answer to that can be used to make the robot move:

| | |
|---|---|
| `#ev3.bev.drive(poke,heading(30))` | set target heading |
| `#ev3.bev.drive(poke,pwlevel(0.5))` | set power level |
| `#ev3.bev.drive(peek,position(:l))` | get the estimated position of the robot (using odometry) |
| `#ev3.bev.drive(poke,position([0,0]))` | set the estimated position of the robot (using odometry) |
| `#ev3.bev.drive(call,move)` | move towards the target heading |
| `#ev3.bev.drive(call,turn.to(45))` | stay put but rotate to face the target heading (45 degrees) |
| `#ev3.bev.drive(call,turn.by(-25))` | stay put but rotate to face an offset from the current heading |
| `#ev3.bev.drive(halt)` | stop, don't move nor rotate |

When the *elemental* is executing one of the called functions (`move`, `turn.to` or `turn.by`), it will frequently publish a *statement* to indicate the status of the function. Using it as a trigger *predicate* allows another *elemental* to react to it, for instance, to execute an action after the robot has turned toward a given direction.

Since we have simply added the *elemental* to an existing *fizz* file we do not have to modify our solution file. In the example that follows, we will get the robot moving forward, and then stop it:

```
robot@ev3dev:~/fizz.0.6.0-X  ./fizz.ev3 ./etc/ev3/robot.json
fizz 0.6.0-X (20190601.1943) [lnx.ev3|1]
Press the ESC key at anytime for input prompt

load : loading ./etc/ev3/robot.json ...
load : loaded ./mod/lnx/ev3/modEV3.so in 0.147s
load : loaded ./mod/lnx/ev3/modCLU.so in 0.071s
load : loading ./etc/ev3/system.fizz ...
load : loaded ./etc/ev3/system.fizz in 0.314s
load : loading ./etc/ev3/sensors.fizz ...
ev3.sen.touch : sensor detected!
ev3.sen.color : sensor detected!
ev3.sen.sonic : sensor detected!
load : loaded ./etc/ev3/sensors.fizz in 0.935s
load : loading ./etc/ev3/motors.fizz ...
ev3.sen.gyros : sensor detected!
ev3.act.motor.l : motor detected!
ev3.act.motor.r : motor detected!
load : loaded ./etc/ev3/motors.fizz in 0.446s
ev3.act.motor.t : motor detected!
load : loading ./etc/ev3/ticks.fizz ...
load : loaded ./etc/ev3/ticks.fizz in 0.102s
load : loading ./etc/ev3/heartbeat.fizz ...
load : loaded ./etc/ev3/heartbeat.fizz in 0.406s
load : loading ./etc/ev3/behaviors.fizz ...
load : loaded ./etc/ev3/behaviors.fizz in 1.285s
load : loading ./etc/ev3/network.fizz ...
load : loaded ./etc/ev3/network.fizz in 0.750s
load : loading completed in 5.156s
?- /spy(append,ev3.bev.state)
spy : observing ev3.bev.state
?- /spy(append,ev3.bev.drive)
spy : observing ev3.bev.drive
?- #ev3.bev.drive(poke,pwlevel(0.5))
spy : Q #ev3.bev.drive(poke, pwlevel(0.500000)) (14.994295)
spy : R ev3.bev.drive(poke, pwlevel(0.500000)) (14.975756)
-> (  ) := 1.00 (0.071) 1
?-
spy : S ev3.bev.state() := 0.00 (14.978130)
spy : S ev3.bev.state(hint, power(on)) (15.000000)
?- #ev3.bev.drive(call,move)
spy : Q #ev3.bev.drive(call, move) (14.992318)
spy : R ev3.bev.drive(call, move) (14.965857)
-> (  ) := 1.00 (0.067) 1
spy : S ev3.bev.drive(hint, move(0, [0.016133, 0], 0)) (15.000000)
spy : S ev3.bev.drive(hint, move(0, [0.045956, 0], 0)) (15.000000)
spy : S ev3.bev.drive(hint, move(0, [0.075288, 0.000158], 0)) (15.000000)
spy : S ev3.bev.drive(hint, move(0, [0.104621, 0.000158], 0)) (15.000000)
spy : S ev3.bev.drive(hint, move(1, [0.134929, 0.000503], -1)) (15.000000)
spy : S ev3.bev.drive(hint, move(2, [0.164005, 0.001339], -2)) (15.000000)
spy : S ev3.bev.drive(hint, move(2, [0.193565, 0.002372], -2)) (15.000000)
spy : S ev3.bev.drive(hint, move(2, [0.223124, 0.003404], -2)) (15.000000)
spy : S ev3.bev.drive(hint, move(1, [0.252938, 0.004091], -1)) (15.000000)
spy : S ev3.bev.drive(hint, move(1, [0.281533, 0.004590], -1)) (15.000000)
spy : T ev3.bev.state
spy : S ev3.bev.drive(hint, move(2, [0.311838, 0.005187], -2)) (15.000000)
?- #ev3.bev.drive(halt,move)
spy : S ev3.bev.drive(hint, move(2, [0.340909, 0.006202], -2)) (15.000000)
spy : S ev3.bev.drive(hint, move(2, [0.370225, 0.007226], -2)) (15.000000)
spy : S ev3.bev.drive(hint, move(2, [0.400273, 0.008275], -2)) (15.000000)
spy : S ev3.bev.drive(hint, move(2, [0.428611, 0.009265], -2)) (15.000000)
spy : S ev3.bev.drive(hint, move(1, [0.458180, 0.009947], -1)) (15.000000)
spy : S ev3.bev.drive(hint, move(2, [0.487994, 0.010613], -2)) (15.000000)
spy : S ev3.bev.drive(hint, move(2, [0.517554, 0.011645], -2)) (15.000000)
spy : S ev3.bev.drive(hint, move(1, [0.547118, 0.012503], -1)) (15.000000)
spy : S ev3.bev.drive(hint, move(1, [0.575958, 0.013006], -1)) (15.000000)
spy : S ev3.bev.drive(hint, move(1, [0.605779, 0.013185], -1)) (15.000000)
spy : S ev3.bev.drive(hint, move(0, [0.634376, 0.013522], 0)) (15.000000)
spy : S ev3.bev.drive(hint, move(1, [0.665173, 0.013876], -1)) (15.000000)
spy : Q #ev3.bev.drive(halt, move) (14.983712)
spy : S ev3.bev.drive(hint, move(1, [0.695479, 0.014405], -1)) (15.000000)
spy : S ev3.bev.drive(hint, move(1, [0.695479, 0.014405])) (15.000000)
spy : R ev3.bev.drive(halt, move) (14.952221)
```

```
-> (  ) := 1.00 (0.127) 1
```

## Sonar behavior

The second more advanced component we are now going to add is a *sonar* which, given an ultrasonic sensor mounted on a motor, can be used to get a sense of what is around our robot. For performance reasons, the `CLU` module provides an *elemental* class (`EV3CBEVSonar`) that implements this.

Re-open once more the `behaviors.fizz` file and copy into it the definition that follow. Here again, refer to the *fizz* user manual for explanations on the properties beyond the comments left in the code:

```
ev3.bev.sonar {

    class   = EV3CBEVSonar,
    chatty  = yes,

    gyros = ev3.sen.gyros,      // label of the gyros sensor (optional)
    sonic = ev3.sen.sonic,      // label of the sonic sensor
    motor = ev3.act.motor.t,    // label of the motor
    drive = ev3.bev.drive,      // label of the drive behavior (optional)

    scan.mtime = 250,           // how often to check if the motor has reached the target position (in ms)
    scan.itime =  50,           // how long after a step before reading the sonic sensor (in ms)
    scan.speed = 270,           // speed of the motor to be applied in scan mode
    skim.mtime = 250,           // how often to read from the sensor while the motor is turning in skim mode (in ms)
    skim.speed = 80             // speed of the motor to be applied in skim mode

} {}
```

As you can see, the *elemental* ties up four of the *elementals* we have defined so far. For the scope of this article, we are not going to dive into each of the properties. Here again, if you have used different names for the *elementals* we have created in this article, you will need to update the values of the `gyros`, `sonic`, `motor` and `drive` properties.

Just like the previous *elemental* we have added, this one will answer to *predicates*. Here are some examples:

| | |
|---|---|
| `#ev3.bev.sonar(call,scan([-90,-45,0,45,90]))` | scan by moving the sensor sequentially to 5 different relative orientations. |
| `#ev3.bev.sonar(call,scan.max([-90,-45,0,45,90]))` | scan by moving the sensor sequentially to 5 different relative orientations, and provides the direction in which the distance is the largest |
| `#ev3.bev.sonar(call,scan.min([-90,-45,0,45,90]))` | scan by moving the sensor sequentially to 5 different relative orientations, and provides the direction in which the distance is the smallest |
| `#ev3.bev.sonar(call,skim([-20,20]))` | scan by moving the sensor in one slow continuous motion in between two relative orientations. |
| `#ev3.bev.sonar(call,skim.max([-20,20]))` | scan by moving the sensor in one slow continuous motion in between two relative orientations and provides the direction in which the distance is the largest. |
| `#ev3.bev.sonar(call,skim.min([-20,20]))` | scan by moving the sensor in one slow continuous motion in between two relative orientations and provides the direction in which the distance is the smallest. |
| `#ev3.bev.sonar(halt)` | halt any scanning and returns to the relative orientation of 0. |

When the *elemental* has completed executing one of the called functions, it will publish a *statement* that will contain the result. Using it as a trigger *predicate* allows another *elemental* to react to it, for instance, to turn the robot into the direction with the clearest path.

Since we have simply added the *elemental* to an existing *fizz* file we can here again try without having to modify our solution file:

```
?- /spy(append,ev3.bev.sonar)
spy : observing ev3.bev.sonar
?- #ev3.bev.sonar(call,scan([-90,-45,0,45,90]))
spy : Q #ev3.bev.sonar(call, scan([-90, -45, 0, 45, 90])) (14.992524)
spy : R ev3.bev.sonar(call, scan([-90, -45, 0, 45, 90])) (14.952157)
-> (  ) := 1.00 (0.117) 1
spy : S ev3.bev.sonar(hint, scan(1560136694.709000, [[-88, 2.001000, [0.695479, 0.014405]], [-44, 2.550000, [0.695479,
     0.014405]], [1, 2.335000, [0.695479, 0.014405]], [43, 0.864000, [0.695479, 0.014405]], [88, 0.786000, [0.695479,
     0.014405]]])) (15.000000)
?- #ev3.bev.sonar(call,scan.max([-90,-45,0,45,90]))
spy : Q #ev3.bev.sonar(call, scan.max([-90, -45, 0, 45, 90])) (14.992260)
spy : R ev3.bev.sonar(call, scan.max([-90, -45, 0, 45, 90])) (14.974081)
-> (  ) := 1.00 (0.071) 1
spy : S ev3.bev.sonar(hint, scan.max(1560136711.072965, [[-88, 1.989000, [0.695479, 0.014405]], [-44, 2.550000, [0.695479,
     0.014405]], [-1, 2.550000, [0.695479, 0.014405]], [43, 0.864000, [0.695479, 0.014405]], [88, 0.786000, [0.695479,
     0.014405]]], [-44, 2.550000, [0.695479, 0.014405]])) (15.000000)
```

Each `hint` *statement* the *elemental* publishes will contains the *list* of readings. Each reading will itself be a *list* containing the absolute orientation at which the reading was taken, and followed by the measured distance (in meters). The third *term* will be the position of the robot at the time of the reading, as estimated by the odometry. In the case of the `scan.max` function, the *statement* will contain a third *term* that will be a copy of the reading with the largest distance. If the function was `scan.min`, it will be the reading with the smallest distance.

## Sensing behavior

The third and last advanced component we are now going to add is a *sensing* one which will combine readings from sensors and motors into a single time-stamped *statement* that will get published with a given frequency. Here again, for performance reasons the `CLU` module provides that *elemental* as a class (`EV3CBEVSense`).

Re-open once more the `behaviors.fizz` file and copy into it the definition that follows:

```
ev3.bev.sense {

    class = EV3CBEVSense,
    ticks = 250,
    terms = [
        [ev3.sen.color,value],
        [ev3.sen.sonic,value],
        [ev3.act.motor.t,position],
        [ev3.sen.gyros,value],
        [ev3.bev.drive,position]
    ],
    mode  = auto

} {}
```

The property `terms` is the list of all the sensors or motors that we wish to include in the *statement*. After the label of the *elemental* to be queried, the name of the property to be fetched is expected. Each of the *elementals* is expected to answer to `peek` *predicate* such as `#ev3.sen.gyros(peek,value(:v))`. The order in which the *elementals* are presented in the `list` will define the order in which their values will show-up in the *statements* second term. The `ticks` property indicates how often (in ms) we want the referenced *elementals* to be queried in the automatic `mode`. Which is the mode in which we will be using the *elemental* for this robot. Note, that a *statement* will only be published if at least one of the values coming from the sensors or motors have changed.

Let's try this out:

```
robot@ev3dev:~/fizz.0.6.0-X  ./fizz.ev3 ./etc/ev3/robot.json
fizz 0.6.0-X (20190601.1943) [lnx.ev3|1]
Press the ESC key at anytime for input prompt

load : loading ./etc/ev3/robot.json ...
load : loaded ./mod/lnx/ev3/modEV3.so in 0.082s
load : loaded ./mod/lnx/ev3/modCLU.so in 0.022s
load : loading ./etc/ev3/system.fizz ...
load : loaded ./etc/ev3/system.fizz in 0.183s
load : loading ./etc/ev3/sensors.fizz ...
ev3.sen.touch : sensor detected!
ev3.sen.color : sensor detected!
ev3.sen.sonic : sensor detected!
load : loaded ./etc/ev3/sensors.fizz in 0.524s
load : loading ./etc/ev3/motors.fizz ...
ev3.sen.gyros : sensor detected!
ev3.act.motor.l : motor detected!
ev3.act.motor.r : motor detected!
load : loaded ./etc/ev3/motors.fizz in 0.454s
ev3.act.motor.t : motor detected!
load : loading ./etc/ev3/ticks.fizz ...
load : loaded ./etc/ev3/ticks.fizz in 0.125s
load : loading ./etc/ev3/heartbeat.fizz ...
load : loaded ./etc/ev3/heartbeat.fizz in 0.451s
load : loading ./etc/ev3/behaviors.fizz ...
load : loaded ./etc/ev3/behaviors.fizz in 0.993s
load : loading ./etc/ev3/network.fizz ...
load : loaded ./etc/ev3/network.fizz in 0.420s
load : loading completed in 3.820s
?- /spy(append,ev3.bev.sense)
spy : observing ev3.bev.sense
spy : S ev3.bev.sense(hint, scan(1560138582.684444, [0, 1.762000, 0, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138582.930861, [0, 1.763000, 0, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138583.430375, [0, 1.766000, 0, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138583.680716, [0, 1.762000, 0, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138583.930339, [0, 1.763000, 0, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138584.680668, [0, 1.764000, 0, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138584.929949, [0, 1.763000, 0, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138585.184008, [0, 1.762000, 0, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138585.430722, [0, 1.763000, 0, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138585.930447, [0, 1.764000, 0, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138586.180119, [0, 1.763000, 0, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138586.680684, [0, 1.767000, 0, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138586.930651, [0, 1.764000, 0, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138587.180854, [0, 1.763000, 0, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138588.180581, [0, 1.762000, 0, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138588.430206, [0, 1.763000, 0, 0, [0, 0]])) (15.000000)
?- #ev3.bev.sonar(call,scan.max([-90,-45,0,45,90]))
spy : S ev3.bev.sense(hint, scan(1560138593.181025, [0, 1.762000, 0, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138593.434390, [0, 1.763000, 0, 0, [0, 0]])) (15.000000)
-> (  ) := 1.00 (0.057) 1
spy : S ev3.bev.sense(hint, scan(1560138593.680864, [0, 1.679000, -28, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138593.930725, [0, 0.429000, -92, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138594.180544, [0, 0.464000, -70, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138594.430978, [0, 1.787000, -28, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138594.684519, [0, 1.754000, 11, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138594.931040, [0, 0.863000, 43, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138595.191754, [0, 0.856000, 52, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138595.430757, [0, 0.780000, 88, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138595.684543, [0, 0.780000, 71, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138595.930399, [0, 2.372000, 7, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138596.180940, [0, 1.764000, 0, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138596.429753, [0, 1.767000, 0, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138596.679725, [0, 1.764000, 0, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138596.930038, [0, 1.767000, 0, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138597.179934, [0, 1.764000, 0, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138598.430492, [0, 1.763000, 0, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138598.680638, [0, 1.764000, 0, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138598.930253, [0, 1.762000, 0, 0, [0, 0]])) (15.000000)
spy : S ev3.bev.sense(hint, scan(1560138599.180388, [0, 1.767000, 0, 0, [0, 0]])) (15.000000)
```

Note, how how the second and third *terms* in the *list* change after a scan is requested from the sonar.

Before moving on to the last section of this article, we need to go back to the `network.fizz` file and add the last few *elementals* we have created to the `filter` list so that they can be accessible from a remote computer:

```
    filters     =   [
                        ev3.sys,
                        ev3.act.motor.t, ev3.act.motor.l, ev3.act.motor.r,
                        ev3.sen.color, ev3.sen.sonic, ev3.sen.gyros,
                        ev3.bev.state, ev3.bev.sense, ev3.bev.sonar, ev3.bev.drive
                    ],
```

## Autonomous exploring

Now that we have all the components in place, we are ready to combine them all to make our robot autonomously wander around in a space. I have used the term *behavior* earlier to categorize some of the *elementals* we are using. In the case of the *elemental* we are building in this section, I will be using the term *instinct* as it has a higher level of complexity and it is built on top of *behaviors*.

The simple procedure that the robot will be following is going to be the following:

1. Use the *sonar* to find a direction in which to head (the one where the reading is the largest distance)
2. Turn (in-place) to face the direction
3. Move (in that direction)
4. If an obstacle is detected with the *sonar* in about the direction the robot is heading, stop and go to 1
5. If an obstacle is detected with the *Color sensor*, stop and go to 1

The robot will also respect its *power state* as we have setup earlier. When the *state* is set to `off`, we will have the robot stop as soon as possible if it is moving or turning.

To start, create a new *fizz* file called `instincts.fizz` and copy into it the following definition for our new *elemental*:

```
ev3.ins.xplorer {

    chatty                  = no,
    replies.are.triggers    = no,

} {}
```

The `chatty` and `replies.are.triggers` properties we have set are only needed here due to the performances constraints of the *EV3*. They basically ensure that the amount of unnecessary inferences will be kept to the strict minimum.

Let's now modify our solution file to include the new *fizz* file:

```
{
    "solution" : {
        "modules" :   ["modEV3","modCLU"],
        "sources" :   [ "system.fizz","sensors.fizz","motors.fizz","ticks.fizz","heartbeat.fizz",
                        "behaviors.fizz","network.fizz","instincts.fizz"],
        "globals" :   []
    }
}
```

From the procedure we have written down earlier, we can see that there are four different active states the robot can be in: idle, moving, turning and picking where to head next. To keep track of what state we're in, we are going to add a property to the *elemental* calling it `state`. The *symbols* we will use to represent each of the states we have listed above are: `null`, `move`, `turn` and `pick`.

We are also going to define two properties (`wide.scan` and `wide.scan2`) to provide the list of relative angles we wish the sonar to mesure when picking a new direction to head out to. The second *list* will be using when its the *Color sensor* that triggered a *stop*. When this occurs, we can't possibly continue moving forward and thus we won't consider that direction when scanning.

When the robot is moving, we are going to use the *sonar* to scan ahead of the robot. But instead of a wide scanning, we will perform a much more narrow scan. To tune this up, we are going to use the property `skim.scan` to provide the list of relative angles to be scanned. We'll also repeat the scan while the robot is moving with a set time interval between consecutive scans. The property `skim.delay` will provide that value (in milliseconds).

Lastly, we'll use the properties `turn.speed` and `move.speed` to scale the speed value of the motors. We will also define the `proximity` property to provide the minimum distance (in meters) at which an obstacle becomes an obstacle to be avoided.

We can now update the definition of our *elemental* as follow:

```
ev3.ins.xplorer {

    chatty                  = no,
    replies.are.triggers    = no,

    state                   = null,
    wide.scan               = [-135,-90,-45,0,45,90,135],
    wide.scan2              = [-135,-90,-45,45,-90,135],
    proximity               = 0.5,
    turn.speed              = 0.8,
    move.speed              = 0.35,
    skim.delay              = 750,
    skim.scan               = [0,-5,5]

} {}
```

The main way to *interface* with the *elemental* is going to be via a `call` to a function we will call `go`. Here's the definition of the two *prototypes* that will support calling and halting the function:

```
1       (call,go)^  :-  peek(state,null),
2                       #ev3.bev.state(peek,power(on)),
3                       ~self(step,pick( wide.scan));
4
5       (halt,go)^  :-  peek(state,_?[neq(null)]),
6                       ~self(step,null);
```

When a request to get the function started is made (the predicate unifies with the *prototype*'s entrypoint) on line 1, we first verify that the current `state` is `null` (e.g. we are not already executing the function) using the `peek` *primitive*. We then validate that the robot is currently powered ON by making a query to `ev3.bev.state` (on line 2). If the *Touch sensor* we are using as a power button hasn't been pressed, the query will fail which mean the inference will fail and we won't get to line 3. If the robot is powered, we will query the *elemental* itself to execute the first of the steps we described earlier (`pick`) using the `wide.scan` property we set recently. The second *prototype* (on line 5) handles the request to stop the `go` function. It first ensures that the current state of the *elemental* is not `null` then queries itself to get the internal state of the *elemental* to switch to `null`.

Before we move onto the actual implementation of each of the states, it is worth noting that *fizz* (at the time of this writing) doesn't support preventing some *prototype* from being queried by another *elemental*. Thus, it is possible to query directly the *prototypes* (which we will consider to be *private*) that we are about to add without going thru the the one we have setup and consider to be *public*.

Now, the `pick` state's purpose, as we have seen earlier, is to use the *sonar* to find a direction to head towards when the robot is stationary. We are going to implement this with the following *prototype*:

```
1    (step,pick(:s))^    :-  console.puts(step.pick(:s)),
2                            peek(state,_?[neq(pick)]),
3                            poke(state,pick),
4                            #ev3.bev.sonar(call,scan.max(:s));
```

Once the query gets matched with the entrypoint of the *prototype* and the *variable* `s` gets assigned with the *list* of relative angles we want to scan for obstacles, we then use the *primitive* `console.puts` to print on the console some tracing information. We then ensure that the current state of the *elemental* is not the one we are trying to set. Once this is done, the inference will move to (line 3) setting the state of the *elemental* to `pick` before calling the `scan.max` function of the *sonar* behavior with the requested *list* of angles (`s`).

As we discussed before, the query to `ev3.bev.sonar` will be answered as soon as the requested function starts. For the *elemental* to be made aware of the result of the *sonar* scan, we are going to use a trigger based *prototype*. In fact, we are going to have to add two *prototypes* as there is two possible outcomes of the scan: the furthest obstacle is further away than the `proximity` value we have setup in the properties or it is closer.

Here is the definition of the *prototype* that will handle the first case:

```
1    ()                  :-  @ev3.bev.sonar(hint,scan.max(_,_,[:h,:d,_])),
2                            peek(state,pick),
3                            gt(:d, proximity)^,
4                            ~self(step,turn.to(:h)),
5                            hush;
```

On line 1, we will get the heading (which is the absolute heading, not the relative heading) (*variable* `h`) and distance (*variable* `d`) by unification of the *statement* published by `ev3.bev.sonar` with our *predicate*. We then check that we are (still) in the `pick` state before (in line 3) using the *primitive* `gt` to ensure that the distance is greater than the value of the `proximity` property. Because we are going to have more than a single *prototype* getting activated by the same *statement*, we postfix the *primitive* call by the *cut* symbol. This will ensure that if the inference goes beyond that point, none of the other concurrent inferences triggered by the same *statement* will continue (or execute at all). If the distance detected by the *sonar* satisfies the constraint, we query the *elemental* itself to execute the next step (`turn.to`) which we will define later.

When the robot ends-up in a place where every possible direction is closer than what we feel comfortable getting close too, we will get the robot to turn around and face a random direction:

```
1    ()                  :-  @ev3.bev.sonar(hint,scan.max(_,_,[:h,_,_])),
2                            peek(state,pick),
3                            rnd.sint(1,:v,-10,+10), add(180,:v,:a),
4                            #add.angle(:h,:a,:nh),
5                            ~self(step,turn.to(:nh)),
6                            hush;
```

This is accomplished on line 3 by picking a random number between `-10` and `10` (using the *primitive* `rnd.sint`) and then adding it to 180 (*primitive* `add`) before using it as argument to the `step` we wish the *elemental* to execute now (line 5). As angles provided by the *gyroscope sensor* are expressed in degrees from `-180` to `180`, we need to insure that the heading we want our robot to turn to is compatible, thus we query the *procedural knowledge* `add.angle` defined below before passing the result to the `turn.to` step:

```
 1 add.angle {
 2
 3     (:a,:b,:c)  :- add(:a,:b,:ab), ~fix.angle(:ab,:c);
 4 }
 5
 6 fix.angle {
 7
 8     (:a?[<0|180>],:a)^  :- true;
 9     (:a?[<-180|0>],:a)^ :- true;
10     (:a?[lt(0)],:b)^    :- mod(:a,360,:a2), add(360,:a2,:b);
11     (:a?[gt(180)],:b)^  :- mod(:a,360,:a2), sub(:a2,360,:b);
12
13 }
```

The *procedural knowledge* we defined here is pretty straightforward: `add.angle` first numerically add the two angle values, then query `fix.angle` to ensure that the sum of the angles stays between the expected bounds. This is implemented by having the *elemental* `fix.angle` pick up the right *prototype* during the unification of the *prototypes*' entrypoint. Note the use of *variable's constraints* (e.g. `:a?[<0|180>]`) to minimize the runtime cost of each *prototype* by making any constraint part of the unification.

Let's now look at how we would implement the prototype dealing with turning the robot towards a given (absolute) heading:

```
1     (step,turn.to(:h))^ :-  console.puts(step.turn.to(:h)),
2                             peek(state,pick),
3                             poke(state,turn.to),
4                             ~self(exec,turn.to(:h));
```

Simarly to the previous related *prototype*, we start with a tracing the inference then check that the current `state` of the *elemental* is indeed `pick`, before changing it to `turn.to` with the *primitive* `poke`. The *prototype* concludes with a query that will execute the turn which we will define as follows:

```
1     (exec,turn.to(:h))^ :-  console.puts(exec(turn.to(:h))),
2                             #ev3.bev.drive(poke,[pwlevel( turn.speed)]),
3                             #ev3.bev.drive(call,turn.to(:h));
```

Lines 2 and 3 query the *elemental* `ev3.bev.drive` to first set the power level to our `turn.speed` property before calling the `turn.to` function. As for the *sonar* function call we made when scanning for a place to head towards, the query will complete as soon as the robot starts to move. To know when the turn is complete, so that we can start moving forward, we are going to use another *trigger prototype*:

```
1     ()                  :-  @ev3.bev.drive(hint,turn.to(:a)),
2                             peek(state,turn.to),
3                             mao.abs(:a,:a.abs?[lte(1)]),
4                             ~self(step,move),
5                             hush;
```

Unlike the *sonar elemental*, the *drive elemental* will publish `ev3.bev.drive` *statements* frequently while executing the `turn.to` function. For each such *statement*, the *term* in the *functor* `turn.to` will be the difference between the current heading and the target heading. We will use the *primitive* `mao.abs` on line 3 to ensure that the inference triggered by the *statement* only continues past that *predicate* when that value is less or equal to `1` degree. When the robot has turned towards the desired heading, the *elemental* will query itself to change its state to `move`. We will define the *prototype* for that as follow:

```
1     (step,move)^    :-  console.puts(step.move),
2                         peek(state,_?[neq(move)]),
3                         poke(state,move),
4                         ~self(exec,move),
5                         #ev3.bev.sonar(call,scan.min( skim.scan, skim.delay));
```

Here again, the *prototype* outputs some trace to the console before changing the `state` property to `move` only if it isn't already the current `state`. The inference then continues by querying the *elemental* to get the robot moving before querying `ev3.bev.sonar` to execute the `scan.min` function. Unlike the other times we have called a function from the *sonar*, the *predicate* this time provides, as second *term*, a delay (in miliseconds) we want the function to be repeated at. As long as the robot is moving, we want the *sonar* to continue performing a narrow scanning of what is ahead of the robot. Providing that optional *term* to the function will ensure that the *elemental* keeps executing the function every so often without having to explicitly call the function over and over.

Let's now have a quick look at the definition of the `move` *prototype*. No surprise here, lines 2 and 3 query the *elemental* `ev3.bev.drive` to first set the power level to our `move.speed` property before calling the `move` function:

```
1    (exec,move)^        :-  console.puts(exec(move)),
2                            #ev3.bev.drive(poke,pwlevel( move.speed)),
3                            #ev3.bev.drive(call,move);
```

While the robot is moving, we need to inspect the result of the `scan.min` function we have requested the *sonar* to be executing. When the value is below the `proximity` property we have set, we will want the robot to stop immediately and try to pick a new direction to head towards. If the distance is greater than `proximity`, we will just output a trace on the console. The two following *prototypes* define these:

```
1    ()                  :-  @ev3.bev.sonar(hint,scan.min(_,_,[:h,:d,_])),
2                            peek(state,move),
3                            lte(:d, proximity)^,
4                            console.puts("proximity ",sonar(:d),"!"),
5                            ~self(step,stop),
6                            hush;
7
8    ()                  :-  @ev3.bev.sonar(hint,scan.min(_,_,[:h,:d,_])),
9                            peek(state,move),
10                           gt(:d, proximity)^,
11                           console.puts("proximity ",sonar(:d)),
12                           hush;
```

The *prototype* that will handle the query to change the *state* to `stop` is defined as follow:

```
1    (step,stop)^    :-  console.puts(step.stop),
2                        poke(state,null),
3                        ~self(exec,stop),
4                        ~self(step,pick( wide.scan));
```

It sets the `state` to `null` on line 2, before executing the actual stop. Then, it queries the *elemental* itself to get the robot to pick a new direction. The prototype executing the stop is also pretty straightforward. It requests both the *drive* amd *sonar* behaviors to stop the execution of the functions they are running. In the case of `ev3.bev.drive`, this will cause the robot motion to stop:

```
1    (exec,stop)^        :-  console.puts(exec(stop)),
2                            #ev3.bev.drive(halt),
3                            #ev3.bev.sonar(halt);
```

There are two more *trigger prototypes* we need to add to have a complete *autonomous* system. The first one will deal with the reading from the *Color sensor*. As you may recall, the value we read from the sensor is provided as one of the *terms* in the `scan` *functor* of the *statements* that are published by the `ev3.bev.sense`. Knowing that the value will jump from 0 to anything less than 1 when the sensor is a few centimeters aways from a surface, we can write it as follow:

```
    ()                 :-  @ev3.bev.sense(hint,scan(_,[:c?[gt(0)]|_])),
                           peek(state,move), console.puts("proximity alert! ",color(:c)),
                           ~self(step,stop.c),
                           hush;
```

Here also, we use a *variable constraint* to express that the *trigger predicate* should only unify when the value from the *Color sensor* is greater than `0`. If this happens when the robot is moving (the `state` of the *elemental* will be `move`), we will output a trace message on the console then query the *elemental* itself. The *prototype* for that will be setting the `state` property to `null` then executing a stop before moving back to the `pick` step. Note that since this stop was originated by the *Color sensor* that is facing forward, we will look for a new direction skipping the forward direction (by using the *list* in `wide.scan2`):

```
    (step,stop.c)^  :-  console.puts(step.stop.c),
                        poke(state,null),
                        ~self(exec,stop),
                        ~self(step,pick( wide.scan2));
```

The second, and last *trigger based prototype* we need to add is one handling the robot's power being changed to OFF. Since, the *elemental* `ev3.bev.state` will publish a *hint statement* when this occurs, we use it as a *trigger predicate* to set the internal `state` of the *elemental* to `null`:

```
    ()                 :-  @ev3.bev.state(hint,power(off)),
                           peek(state,_?[neq(null)]),
                           ~self(step,null),
                           hush;
```

The `(step,null)` self query reference the following *prototype*, which by now should be straightforward to follow:

```
    (step,null)^    :-  console.puts(step.null),
                        poke(state,null),
                        ~self(exec,stop);
```

Once you have added the last *prototype* to the definition of `ev3.ins.xplorer`, we are ready to give this a try by running it on a computer with the *EV3 Intelligent brick* will be running the rest. First, copy `host.json` into a new file called `host+instincts.json` and add the `instincts.fizz` to it:

```
{
    "solution" : {
        "modules" :   ["modCLU"],
        "sources" :   ["network.fizz","instincts.fizz"],
        "globals" :   []
    }
}
```

We can then start *fizz* on the *EV3* like we did earlier:

```
robot@ev3dev:~/fizz.0.6.0-X  ./fizz.ev3 ./etc/ev3/robot.json
fizz 0.6.0-X (20190601.1943) [lnx.ev3|1]
Press the ESC key at anytime for input prompt
```

```
load : loading ./etc/ev3/robot.json ...
load : loaded ./mod/lnx/ev3/modEV3.so in 0.082s
load : loaded ./mod/lnx/ev3/modCLU.so in 0.022s
load : loading ./etc/ev3/system.fizz ...
load : loaded ./etc/ev3/system.fizz in 0.183s
load : loading ./etc/ev3/sensors.fizz ...
ev3.sen.touch : sensor detected!
ev3.sen.color : sensor detected!
ev3.sen.sonic : sensor detected!
load : loaded ./etc/ev3/sensors.fizz in 0.524s
load : loading ./etc/ev3/motors.fizz ...
ev3.sen.gyros : sensor detected!
ev3.act.motor.l : motor detected!
ev3.act.motor.r : motor detected!
load : loaded ./etc/ev3/motors.fizz in 0.454s
ev3.act.motor.t : motor detected!
load : loading ./etc/ev3/ticks.fizz ...
load : loaded ./etc/ev3/ticks.fizz in 0.125s
load : loading ./etc/ev3/heartbeat.fizz ...
load : loaded ./etc/ev3/heartbeat.fizz in 0.451s
load : loading ./etc/ev3/behaviors.fizz ...
load : loaded ./etc/ev3/behaviors.fizz in 0.993s
load : loading ./etc/ev3/network.fizz ...
load : loaded ./etc/ev3/network.fizz in 0.420s
load : loading completed in 3.820s
```

and then run an instance on a PC on the same network. Once the robot's is powered by the pressing the *Touch sensor*, we can query `ev3.ins.ezplorer` and get the robot moving around:

```
jlv@akkala:~/Code/okb/apps/fizz  ./fizz.x64 ./etc/experiments/ev3/article/host+instincts.json
fizz 0.6.0-X (20190601.2228) [lnx.x64|8|l]
Press the ESC key at anytime for input prompt

load : loading ./etc/experiments/ev3/article/host+instincts.json ...
load : loaded ./mod/lnx/x64/modCLU.so in 0.000s
load : loading ./etc/experiments/ev3/article/network.fizz ...
load : loading ./etc/experiments/ev3/article/instincts.fizz ...
load : loaded ./etc/experiments/ev3/article/network.fizz in 0.002s
load : loaded ./etc/experiments/ev3/article/instincts.fizz in 0.018s
load : loading completed in 0.019s
?- #ev3.ins.xplorer(call,go)
step.pick([-135, -90, -45, 0, 45, 90, 135])
-> (  ) := 1.00 (0.331) 1
step.turn.to(-45)
exec(turn.to(-45))
step.move
exec(move)
proximity sonar(2.550000)
proximity sonar(2.550000)
proximity sonar(2.314000)
proximity sonar(2.261000)
proximity sonar(2.155000)
proximity sonar(2.116000)
proximity sonar(2.072000)
proximity sonar(1.969000)
proximity sonar(1.511000)
proximity sonar(0.864000)
proximity sonar(0.887000)
proximity sonar(0.962000)
proximity sonar(0.879000)
proximity sonar(0.580000)
proximity sonar(1.404000)
proximity sonar(0.380000)!
step.stop
exec(stop)
step.pick([-135, -90, -45, 0, 45, 90, 135])
step.turn.to(-2)
exec(turn.to(-2))
```

If we wanted to run the whole thing on the *EV3*, we'll just have to copy the `robot.json` file into `robot+instincts.json` and add `instincts.fizz` to the list of *knowledge* to be loaded. Then query `ev3.ins.ezplorer` on the *EV3* instance of *fizz* :

```
robot@ev3dev:~/fizz.0.6.0-X  ./fizz.ev3 ./etc/ev3/robot+instincts.json
```

```
fizz 0.6.0-X (20190601.1943) [lnx.ev3|1]
Press the ESC key at anytime for input prompt

load : loading ./etc/ev3/robot+instincts.json ...
load : loaded ./mod/lnx/ev3/modEV3.so in 0.082s
load : loaded ./mod/lnx/ev3/modCLU.so in 0.022s
load : loading ./etc/ev3/system.fizz ...
load : loaded ./etc/ev3/system.fizz in 0.185s
load : loading ./etc/ev3/sensors.fizz ...
ev3.sen.touch : sensor detected!
ev3.sen.color : sensor detected!
ev3.sen.sonic : sensor detected!
load : loaded ./etc/ev3/sensors.fizz in 0.578s
load : loading ./etc/ev3/motors.fizz ...
ev3.sen.gyros : sensor detected!
ev3.act.motor.l : motor detected!
ev3.act.motor.r : motor detected!
load : loaded ./etc/ev3/motors.fizz in 0.448s
ev3.act.motor.t : motor detected!
load : loading ./etc/ev3/ticks.fizz ...
load : loaded ./etc/ev3/ticks.fizz in 0.098s
load : loading ./etc/ev3/heartbeat.fizz ...
load : loaded ./etc/ev3/heartbeat.fizz in 0.391s
load : loading ./etc/ev3/behaviors.fizz ...
load : loaded ./etc/ev3/behaviors.fizz in 1.002s
load : loading ./etc/ev3/network.fizz ...
load : loaded ./etc/ev3/network.fizz in 0.387s
load : loading ./etc/ev3/instincts.fizz ...
load : loaded ./etc/ev3/instincts.fizz in 3.261s
load : loading completed in 7.174s
?- #ev3.ins.xplorer(call,go)
step.pick([-135, -90, -45, 0, 45, 90, 135])
-> (  ) := 1.00 (0.202) 1
step.turn.to(-89)
exec(turn.to(-89))
step.move
exec(move)
proximity sonar(2.123000)
proximity sonar(0.370000)!
step.stop
exec(stop)
step.pick([-135, -90, -45, 0, 45, 90, 135])
step.turn.to(-133)
exec(turn.to(-133))
step.move
exec(move)
proximity sonar(1.227000)
proximity sonar(1.160000)
proximity alert! color(0.010000)
step.stop.c
exec(stop)
step.pick([-135, -90, -45, 45, -90, 135])
step.turn.to(2)
exec(turn.to(2))
step.move
exec(move)
```

## Going further

The example discussed in this document, can serve as the starting point for further exciting experimentations which are outside of the scope of this article. For instance, using *odometry* and the outputs from the *sonar* it would be possible to create a *symbolic map* of the space in which the robot is roaming. With that map and some pathfinding *procedural knowledge*, the robot could be made to head towards particular places on the map.

Another exciting experiment is to turn the robot into a *Conscious Turing Machine* [6] (running the CTM on a PC and not on the *EV3* due to processing constraints) and observe if adaptability arises from it.

[6] http://f1zz.org/downloads/ctm.pdf