Une Language

Francisco Morero Peyrona

Preface

This work was initiated as per my PhD thesis.

It is dedicated to my family: their partners, children, parents and siblings (recursively).

About the mascot

The mascot for the *Une* language is an electric anguilla named *Tune*.

For more information about this amazing animal, visit:
https://en.wikipedia.org/wiki/Electrophorus_electricus

About the names

- **Mingle**
  This is the name given to the whole ecosystem (the platform): the language and all the tools around it.

  mingle /mĭng′gəl/[1]

  1. To mix or bring together in combination.
  2. To be or become mixed or united.
  3. To associate or take part with others.

  Meaning that this platform aims to be the glue for the IoT. All following names, are referred in one or another way to the idea of "Glue".

- **Une**
  This is the name of the language. In Spanish language it means "what joins".

- **Tape**
  Glue can exist in the form of a tape: used mainly when wrapping gifts. This is the name of the transpiler (a special type of compiler). The transpiler transforms and optimizes source *Une* code into its abstract representation.

1 The American Heritage® Dictionary of the English Language, 5th Edition.

- **Stick**
  Glue-sticks are mainly used by kids. It is the name of the Execution Environment: it loads an *Une* code abstract representation and executes it.

- **Glue**
  This is the traditional form of: a viscose liquid. This is the name of the IDE (Integrated Development Environment).

- **Gum**
  This is an ancient form of glue. This is the name of the monitoring tool.

Special thanks

To (in alphabetical order):

- Gabarrón Luque, Anto.
- Martín Gómez, Juan Domingo.

Summary

Chapter 1
Overview

Introduction

"*Une*" is the language for the Mingle Platform. *Une* was designed with simplicity in mind: it is so easy that anyone with basic concepts of a spreadsheet can learn *Une* in one single day, at least its basics, but enough to manage a family house.

Une is very close to a natural language (these languages that we use to communicate with other human beings). Although this handbook uses *Une* in English, it can be "translated" into other languages (Spanish, German, French, Chines, ..); later we will explain how to do it.

A sneak preview

Here is how to say: "when the alarm is on and any door is opened, then send a message to my Telegram[2]".

```
WHEN alarm IS ON AND ANY door IS OPEN
THEN MyTelegram SET "DANGER! Intruders at home"
```

How is this book organized

1. We will start explaining some basic concepts.
2. Next we will go through several examples, increasing the complexity.
3. And we will end with the formal description of the language.

How to learn *Une* in one day

If you are familiar with any spreadsheet (MS-Office Excel, LibreOffice Calc, etc.), then you can learn Une basics -enough to manage a home installation- in one day.

There are lots of examples under the folder "examples". The examples are orderer in a increasing complexity order, being 1 the simplest. These examples is one of the two key pieces you need learn *Une*, the other key pieces is this book.

Une is the language, but to compile and run *Une* commands, you need a compiler and an executor. The compiler makes the optimizations and the translations needed by the executor to be able to execute your commands. These two tools (and other tools)

2 https://telegram.org

are covered in the book titled “The Mingle Standard Platform”. Please, refer to this book to learn how to compile and execute *Une* source code.

This is our recommendation to learn *Une*:

1. Read this book Chapter 1
2. Read and run examples starting the first one, until you feel you do not fully understand the example.
3. Read this book next Chapter.
4. Read and run examples where you left it, until you feel you do not fully understand the example.
5. Go to 3.

One more thing before go: *Une* is case insensitive (case is ignored). In all the cases except when using strings: everything in between double-quotes will preserve its case.

Basic Concepts

A domotics (AKA “house automation”[3]) installation is basically a set of **devices** that are responsible for certain tasks: switching lights on and off, changing the temperature of a room or informing if a door or window is open or closed.

In *Une*, we declare the physical Devices that we have (at home, at office or at a factory) and we also declare how to rule them; for this, we use **rules**. We already have seen a rule in the previous example.

Before continuing:

> In following pages, you’ll find lots of foot notes: this is intended. First read the pages without paying attention to foot notes. When you understand the first example, read it again the pages paying attention to the foot notes.

Devices

A Device is every electronic gadget that can inform about its estate and/or modify its estate. All gadgets that you have lets say at home, makes your “home domotic

3 https://en.wikipedia.org/wiki/Home_automation

installation”. The size of the installation is not really important: it can be just a house, a building, a set of buildings or even a big factory: *Une* can handle it.

Devices are divided into 2 groups: Sensors and Actuators. An example of a **Sensor** is a thermometer, because it is a device that reads values. In this case, a thermometer reads temperature. An anemometer reads wind speed, and an hygrometer reads humidity. We call them a *read-only* device because you can not change their values.

An example of an **Actuator** is a relay, because it is a device which state can be changed: ON (closed) or OFF (open). A light is also an Actuator because we can change its state to be ON or OFF. We can know their estate and we can also change their estate, so we call them a *read-and-write* device.

But to make it simpler, we declare both types (Sensors and Actuators) with the same command keyword: **DEVICE**.

You must name every Device with an unique name. And you must associate every Device with an appropriate Driver. We will explain this with more detail along this section.

Technical hint

Some people (mostly experienced engineers) feel more comfortable using the words SENSOR for a device that is a sensor and the word ACTUATOR for a device that is an actuator, instead of using simply the DEVICE word for both. You can do it by using following USE commands:

```
USE “Sensor” AS “Device”

USE “Actuator” AS “Device”
```

But be careful, because it will be replaced everywhere.

Drivers

A Driver in *Une* is very similar to the concept of a “printer driver”: you need a Driver in order to use your printer. And you will need as many drivers as different printers you have. In the same way, devices need a Driver to be able to do their job. You just need to tell *Une* which Driver to use for every device; but be careful, you must associate the right driver with the right device. If you have an Air Conditioner machine manufactured by a company named SumgSan and another by a company named KinDai, then you will need 2 drivers, one for each brand (and perhaps also the specific model).

Drivers are declared inside the `DEVICE` command: using the clause `DRIVER`. We'll see it later.

Now we will provide some further technical explanations, if you do not feel comfortable with technical details, simply skip it.

Technical hint

Some Drivers work only with Sensors, others only with Actuators and others (less common) with both. We can ask for the value of an Actuator and we can also modify it; but when dealing with sensors, we can only ask for their values, it does not make any sense to change a Sensor's value.

If you are interested in creating your own drivers, then refer to the Drivers section in the book "The Mingle Standard Platform". Note: you will need programming knowledge in at least one of the following languages: Java, Python or JavaScript.

Rules

Meanwhile **Devices** describe the gadgets we have, **Rules** describe how devices have to interact with each others: how a domotics installation has to behave. Rules trigger actions under certain conditions (when certain conditions are satisfied).

Lets revisit the Rule shown at the beginning of this tutorial.

```
WHEN alarm IS ON AND ANY door IS OPEN
THEN MyTelegram SET "DANGER! Intruders at home"
```

We have 2 devices in the `WHEN` clause: "`alarm`" (it is an actuator) and "`door`" (it is a group of sensors[4]). The `WHEN` clause is satisfied only when the "`alarm`" is active (is ON) and any device belonging to the group of "`door`" devices is open.

4 We can define a group of devices and refer to all or any of the devices by the name of the group. We'll see how to deal with group of devices later.

Every time a Device changes its estate, all Rules associated[5] with the Device will be evaluated. And when the result of the evaluation is true (when the `WHEN` clause is satisfied), all actions in the `THEN` clause will be executed (triggered). In our example, there is only one action: a message will be sent to the Device "`MyTelegram`" (an actuator),

First example

OK, lets put together. To do so, we will create, a simple example, step by step. We start by declaring a Device; this Device reads computer's clock[6] [7] [8].

```
DEVICE clock
    DRIVER ClockDriver
        CONFIG
            interval SET 3s
```

We start with the device command, which is defined by the keyword `DEVICE`[9], followed by the name of the device, in our case "`clock`" (it is a sensor). This is an arbitrary name that identifies uniquely every device.[10]

The next line is a clause declaring the `DRIVER` that will handle the `DEVICE`. Driver's name is given not by us but by the driver manufacturer: the driver we use in this example is provided by the MSP (Mingle Standard Platform) and its name is "`ClockDriver`".[11]

5 We say a Device is associated with a Rule when the name of a Device appears in the `WHEN` clause of the Rule.
6 This clock reports the lapsed time since the ExEn was started (in milliseconds)
7 *Une* does not differentiate between uppercase and lowercase.
8 Indentation is optional, but most people likes it.
9 Remember `Device` and `device` will also work)
10 We can give any name to the Device as far as it follows certain simple rules that we will explain later.
11 Once again: "clockdriver" and any other combination of upper and lower letters will also work, but when having more than one word combined, it is quite common to write in upper case the first letter of each word and in lower case the rest of the word, this is known as "CamelCase" (https://en.wikipedia.org/wiki/Camel_case). Other people prefer the use of underscores ("_") to separate words: "clock_driver", "camel_case", "these_are_four_words". It is totally up to you: *Une* is agnostic in regarding this.

The next clause is used to configure the Driver: it uses the word **CONFIG**. Almost every Driver provides one or more parameters to configure it, some of them are mandatory (required) and others are optional, they have a value default[12].

In our case, this Driver Configuration accepts only one parameter, named "`interval`". We set the interval to 3 seconds ("s" stands for seconds, "m" for minutes, "h" for hours… We will talk about it later). This interval means that the value of the internal clock will become the value of the device every 3 seconds[13].

In next command, we declare another Device. To keep things as simple as possible, we will use the "`console`" (it is an actuator), also known as the computer's screen. Therefore, when we change the state of this Device, we are printing in the screen.

In the same way we did with the previous Device, we provide an unique name, not surprising it is "`console`".

```
DEVICE console
    DRIVER OutputDriver
```

The next line after the name, is the **DRIVER** clause and says which one we want to use: "`OuputDriver`" in our case. Even if this driver admits several parameters, none of them are mandatory and the default values are OK for our purposes, so we do not need to **CONFIG** the Driver[14].

Now the third and last command: we need to create a Rule[15]:

```
WHEN clock ABOVE 0
    THEN console SET "Clock value is: " + clock
```

As mentioned previously, rules are evaluated every time the associated Device ("`clock`" in this example) changes its value (every 3 seconds in our case) and when the result of the evaluation is true, all actions in the **THEN** clause will be executed (triggered).

12 The amount, the type and the name of the parameters depend on the manufacturer that created the Driver: you can find all this information by reading Driver declaration (check file: {*home*}/include/common-drivers.une)
13 Every time a Device's value is read, it becomes the value of the Device and every time a Device changes its value, all Rules associated with this Device are informed, so they can act as they are instructed.
14 FYI: the device "clock" is a sensor and device "console" is an actuator.
15 Surprisingly the word **RULE** does not appear: it will be explained later.

In our case, our condition sets that the value of the Device named “`clock`” has to be above 0 (zero), therefore the **`WHEN`** clause will always be satisfied (will be always true). So every 3 seconds the value of the Device “`clock`” will change and it will be printed in the screen.

Lets explain this process in more detail:

1. Every 3 seconds, the Driver “`OutputDriver`” changes its Device’s value (“`clock`”). This Device’s value will become the amount of milliseconds elapsed since the application started. This Device’s Driver **reads** values, it is a *read-only* driver.
2. Every time a device’s value changes, all Rules having the Device that changed in their **`WHEN`** clause, will be notified and they will be evaluated.
3. When Rule **`WHEN`** clause is satisfied, all actions in **`THEN`** clause will be executed. In our example, there is only one action: to set a new value for the Device named “`console`”.
4. When we change Device’s value (the console in our case), the Device notifies its Driver and the Driver will change the physical world: in our case, a message will be printed into the screen. This Device’s Driver **reads** and **writes** values, it is a *read-and-write* driver.
5. The “`console`” device changed its estate, so all rules associated with this device will be notified. But there is no rule having “`console`” in their **`WHEN`** clause, therefore, nothing else happens.

Please, note that the value of the “`clock`” is preceded by the message `“Clock value is: “` and the plus sign. This plus sign concatenates the message with the current value of “`clock`”. So, when we execute these 3 commands[16], we will see in the screen of our computer something similar to this:

```
Clock value is: 15248
Clock value is: 18257
Clock value is: 21263
```

And that is it!

Congratulations, now you know most part of what is needed to create and manage your own domotics installation.

16 To execute Une code, it has to be compiled first, and the resulting file is passed to an ExEcution Environment (ExEn for short). The MSP provides a compiler named “Tape” and an ExEn named “Stick”.

Well… to be totally honest, there is one more thing that is needed: some other files (provided by the MSP) have to be included. But right now, all you need to know is that you have to insert the following line in your script[17]:

```
INCLUDE "file://{*home*}include/standard-includes.une"
```

So, the whole script would looks like this[18] [19]:

```
DEVICE clock
    DRIVER ClockDriver
        CONFIG
            interval SET 3s

DEVICE console
    DRIVER OutputDriver

WHEN clock ABOVE 0
    THEN console SET "Clock value is: " + clock

INCLUDE "file://{*home*}include/standard-includes.une"
```

There is not much more to say about Devices. You basically add as many of them as you need and you associate them with their Driver, that's all[20].

On the other hand, there is much more to say about Rules; they are the key point of *Une* because Rules embed (contain, represent) the logic of the system. So, although we will talk more in the next pages about devices, we will focus in Rules.

Ruling the Rules

Une Rules has the following syntax:

17 A script is a file containing a set of *Une* commands.

18 Remember: there is no special order for the commands inside the script, neither for the clauses inside the command.

19 Although you can place the INCLUDE command at any point, normally it is placed at the end of the script file.

20 There is something else: in the same way Drivers can be configured, Devices can be too. But we will talk about it later.

```
RULE <name>
    WHEN <conditions>
    THEN <actions)>
    IF   <future_conditions>
```

There are clearly 2 clauses that we did not cover yet because they are optional: **RULE** and **IF**. Lets now talk about all clauses.

RULE

This clause (it also can work as a command keyword) is used when you need to give a name to the Rule, and you need to name a Rule only if you need to refer to a Rule from another Rule or from the command **SCRIPT**[21].

It is not frequent to name Rules, so we will cover it later.

IF

This clause is also optional and it contains future conditions: these conditions will be evaluated later in time. To illustrate this, we will take same Rule we used at the beginning of this handbook:

```
WHEN alarm IS ON AND ANY door IS OPEN
    THEN MyTelegram SET "DANGER! Intruders at home"
```

This Rule has a problem: when we arrive home, we have no time to deactivate the alarm. Lets say we set it to ON before going out. Later we come back, we open the door and the alarm is immediately triggered. But what we want is to have 30 seconds to deactivate the alarm. This can be achieved easily adding the following **IF** clause:

```
WHEN alarm IS ON AND ANY door IS OPEN
    THEN MyTelegram SET "DANGER! Intruders at home"
    IF alarm IS ON AFTER 30s
```

21 We will cover SCRIPTs later.

When the `WHEN` clause is satisfied, the action in the `THEN` clause will not be executed until the `IF` clause is satisfied, what could (or could not) occur 30 seconds after the `WHEN` was satisfied.

This would be the sequence in time:

1. The main door is opened.
2. Main door's Device changes its state.
3. The change is propagated, the Rules are informed and they are re-evaluated.
4. The `WHEN` clause is satisfied (it is true) but `THEN` action is not triggered yet.
5. After 30 seconds, the `IF` clause is evaluated. If it is satisfied, the action in clause `THEN` is triggered, otherwise it will not.

THEN

Until now we placed only one action at this clause, but we can have as many actions as we want. We can separate them in two ways:

a) By writing a semicolon at the end of each action:

```
THEN MyTelegram SET "Send to Telegram"; console SET "Send to console"
```

b) By placing each action in a different line (semicolon is not needed in this case). This is the recommended way because produces clearer code:

```
THEN MyTelegram SET "Send to Telegram"
     console    SET "Send to console"
```

<u>Important</u>:

> Once a Rule is satisfied, all its actions will be executed, but there is no guarantee about the execution order: they are sent to the execution queue in the order they appear in the code, but it does not mean they will be finished in same order: those actions that need more time to be accomplished will finish

later even if they appear prior to others that need less time. Take this into account because this can be crucial under some circumstances.

WHEN

As we said, this clause (it also can work as a command keyword) contains the "formula" (we prefer the term "expression") that is evaluated every time a Device changes its state.

This expression is very similar to the formulas we write in spreadsheet's cells: most of what can be written in a spreadsheet formula, can be written in **`WHEN`** and **`IF`** clauses: operators (arithmetic, relational and boolean), parenthesis, etc. If you are an intermediate or advanced spreadsheet user, you already know a lot about *Une* expressions.

Keep in mind that **`WHEN`** and **`IF`** expressions must resolve to a Boolean value: TRUE or FALSE. In other words, the result of evaluating the expression must be a Boolean value.

Expressions are evaluated by an expression-evaluator: MS Excel has its own, LibreOffice Calc has its own and *Une* has its own. What can be done and what can not, is something that will be extensively cover later. As per now, keep in mind this: most part of what you can place in a spreadsheet formula can be placed in **`WHEN`** and **`IF`** expressions; you just change the references to other cells by the names of the Devices.

Later we will see Rules in detail, but now, this is all you need to now.

From now and until the end of this handbook, we will be more technical, but do not panic, it will be easy to be understood.

How are Rules evaluated

This is a crucial point you can not forget: every time a device changes its state, all rules associated with the device that changed (the device is part of the **`WHEN`** clause) will be re-evaluated.

Lets be more precise: when a device changes its state all rules are informed (they receive the device name and the new value). The rule then checks if this device is part of its **`WHEN`** clause, and if this is the case, the expression will be evaluated with the new value (and all other previous values it could have from other devices associated with the same rule). And finally, if the expression resolves to **`TRUE`**, then if there is no **`IF`** clause, all associated actions (**`THEN`** clause) will be triggered. But if there is an `IF` clause, it has to be also satisfied in order for the rule to trigger the actions.

Remember: a rule can be also unconditionally triggered by invoking it from another rule (the former rule has to be previously satisfied) or from the command `SCRIPT`.

Chapter 2
Building Blocks

Commands

Introduction

Une was conceived to be as close to a natural language as possible (English, Spanish or Chinese are examples of natural languages).

A command in *Une* is very similar to what we understand by a command, but not exactly the same. In *Une*, the intention of a command is more declarative than imperative: we declare the devices and the rules that manage the devices.

Each *Une* command is written in its own paragraph[22]. In other words, to separate a command from another, we must write one (or more) empty line. It is very, very important to remember this: new lines and empty lines has a special meaning in *Une*.

Commands are composed of parts, these parts are named “clauses”. Following `RULE` command:

```
RULE rule_alarm
    WHEN ANY door IS OPEN AND alarm IS ON
    THEN MyTelegram SET "DANGER! Intruders at home"
    IF alarm IS ON AFTER 30s
```

Is composed by these clauses:

- `WHEN`
- `THEN`
- `IF`

For clarity sake, it is recommended to use at least one line per clause, although we can write all clauses in the same line:

```
RULE rule_alarm WHEN ANY door IS OPEN AND alarm IS ON THEN MyTelegram
SET "DANGER!  Intruders at home" IF alarm IS ON AFTER 30s
```

22 Taken (not literally) from Oxford dictionary: a paragraph is a group of sentences that fleshes out a single idea and is indicated by a new line.

The order of the commands in the file is not important, neither command's clauses inside a command, as far a you start the command with the keyword that represents it: **DEVICE**, **WHEN** (or **RULE**), etc.

In other words: you do not need to declare elements prior to use them. In this way, **RULE**s can appear prior to the **DEVICE** declaration the **RULE** uses.

Technical hint

Perhaps one of the languages that shares more similarities with *Une* is SQL[23]: which is also a Script (command based), Declarative, Domain Specific language. It was developed by IBM to handle Relational Data Base Management Systems (RDBMS).

Une has just following 6 commands:

Command	Brief description	Used by
DEVICE	Declare physical devices	Normal users
RULE	Declare the system logic	Normal users
INCLUDE	Insert other files into current one (used by the preprocessor)	Normal users
USE	Change words and symbols (used by the preprocessor)	• Advanced users • Manufacturers
SCRIPT	Execute other languages code inside the ExEn	• Advanced users • Manufacturers
DRIVER	Connect SCRIPTs with devices	Manufacturers

Summary

Une is made up of commands, commands are composed by clauses, some clauses are optional and others are not. Clauses can have sub-clauses. Clauses can also have parameters which are mostly optional (most of them have a default value). A file containing a set of commands is named a script file. The order of the commands in the file is not important, neither command's clauses inside a command, as far a you start the command with the keyword that represents it. Other files can be referred from inside a file using the **INCLUDE** command.

23 https://en.wikipedia.org/wiki/SQL

Lets clarify this with an example (note that in *Une*, everything after the '#' symbol is ignored: it is used to write comments):

```
DEVICE clock                          # Command descriptor
    DRIVER ClockDriver                # Clause
        CONFIG                        # SubClause
            interval SET 3s           # Parameter
```

Now we will summarize each command by presenting their syntax and describing them briefly. Later we will explain them in detail using examples. But first, we need to establish few...

Syntax Conventions

- Something between "**<>**" means that it must be replaced by its meaning.
- Something between "**[]**" means it is optional
- Something between "**{}**" means there are several options and one of them must exist. Options are separated by "**|**".
- "**...**" means that previous statement can be repeated as many times as needed.

One more thing: commands can have clauses with variable length arguments. As we just said, this is noted with an ellipsis: "**...**". To separate the arguments, you can use a semicolon (";") or place each argument in a separated line[24] or both:

```
# Semicolon separated parameters
INCLUDE "file-one.une"; "file-two.une"; "file-three.une"

# Each parameter in its own line
INCLUDE "file-one.une"
        "file-two.une"
        "file-three.une"

# Both
INCLUDE "file-one.une";
        "file-two.une";
        "file-three.une"

# It is also valid to start the rest of the command in a new line
```

24 This is the recommended way of doing it.

```
INCLUDE
    "file-one.une"
    "file-two.une"
    "file-three.une"
```

Basic Commands

Following you'll find the three commands that you are familiar with: Devices and Rules.

DEVICES

Syntax:

```
DEVICE <name>
  [INIT <property> = <value> [; ...]]
   DRIVER <driver>
     [CONFIG name = <value> [; ...]]
```

Purpose:

To declare the devices that conform our physical installation.

Clauses:

- `INIT`: initializes device properties. Although different implementatioSyntax Conventionsns of the Mingle Platform can expose different properties, the Standard (MSP) one exposes the following:
    - **Groups**: one or more comma separated group names[25] written between quotes.
      e.g.: `groups = “doors, windows, sensors”`
    - **Value**: Set device's initial value (sat only once: when the script starts). Valid only for Actuators. e.g.: `value SET ON`
    - **Delta**: Sets the minimum variation in physical device that will update device's internal value (AKA Hysteresis[26]); e.g.: `delta SET 0.1`
    - **Downtime**: This property allows to find out devices that are potentiall{y damaged. The idea is simple: if a device that should report its value let say

25 Although it is recommended to use *Une* valid names, you can use an combination of alphabetic characters and number and even special symbols. Following are valid names for groups: a_name, a-name, @name, (00#~99), [name]

26 https://en.wikipedia.org/wiki/Hysteresis

every hour did not report it for 3 hours in a row, most probably it will be damaged.

This is what you have to do to make it work:

1. In "`config.json`" file there are 2 values under "exen" module:
    a) "`downtimed_interval`" : the interval (in seconds) to invoke the internal task that looks for the downtimed devices. Zero means do not do it.
    b) "`downtimed_report_device`" A device's name. This device will change its value holding the downtimed devices list.

2. In the *Une* script, define a device named as `<device_name>` and assign the `CellSetDriver` driver to it.

3. Define "downtime" property for every device you want to monitorize.

4. Declare a Rule having a `WHEN` like this: `NOT <device_name>:isEmpty()` and all actions you want in the `THEN` clause.

5. If you are lost, do not worry, example 15 shows it.

- `DRIVER`: Driver’s name to be used to manage the device.

- `CONFIG`: sets the DRIVER configuration. It depends on the DRIVER specified in the previous clause.

Notes:

Remember: while `INIT` refers to the device, `CONFIG` refers to the Driver.

RULES

Syntax (simplified):

```
[RULE <name>]

   WHEN <conditions>
   THEN <actions>
  [IF <future-conditions>]
  [USE <aliases>]
```

```
Preconditions
    {<device> | [ANY | ALL] <group>} <RelationalOp> <expression>

Actions
    {<script> | <rule> | <device> | <group>} = <expression>
    [AFTER <time_unit>] [; ...]
```

```
Future-conditions
    {<device> | [ANY | ALL] <group>} <RelationalOp> <expression>
    [{AFTER | WITHIN} <time_unit>]

Aliases
    USE <name> AS <expression> [; ...]

All together
    WHEN <precondition> [{AND|OR|NOT} <precondition> ...]
    THEN <action> [; ...]
    IF (<future-condition>) [{AND|OR|NOT} (<future-condition>) ...]
    USE <alias> [; ...]
```

Purpose:

To declare the system logic: how the domotics installation will behave.

Clauses:

- `WHEN`: here we place the preconditions that must be satisfied for the actions to be triggered.
- `THEN`: these are actions (one or more) that will be triggered.
- `IF`: here we place future conditions that must be satisfied for the actions to be triggered. When `IF` exists, all conditions in `WHEN` and in `IF` has to be satisfied for the actions to be triggered. When using logical operators, you must enclose each future-condition in parentheses. It is strongly recommended to always use parentheses to clarify how logical and arithmetical expressions are grouped, but in the case of future-conditions, it is mandatory.
- `USE`: this optional clause provides 3 benefits:
    - Improves legibility: makes `WHEN`, `THEN` and `IF` clauses of `RULE` command easier to be read.
    - Improves performance: every time that a name is used (a name is the left side of the '`AS`'), its value can be accessed without needing to evaluate the expression over and over.
    - Because it is evaluated only once, all occurrences will have same value (this is important when managing values than change over time).

Notes:

- Observe that key word `RULE` (which provides Rule's name) is optional. CIL (Command Incarnation Library) automatically provides an unique name to Rules that has no name.

- Do not confuse this `RULE`’s clause `USE` with the command `USE`. In both cases the keywords are the same because their purpose is the same, but their syntax differs slightly.

- When using `USE` clause, the name given can not be used in other clauses. For instance, if we have a `DEVICE` named ‘A_Name’, and also following clause: `USE a_name AS date:date(),` following clause will trigger an error: `WHEN` A_Name `ABOVE 38`

- `USE` clause’s name does not allow the use of *Une* reserved words neither the use of standard functions (except obviously inside strings).

- As Rules are the cornerstone of the *Une* language, we will dedicate the next chapter to them.

COMMENTS

Although comments are not commands, there are few things more useful in computer languages than comments: we, humans need to write notes not to forget things.

In *Une* language, comments start by the sharp (‘#’) sign: whatever we write after it, will be ignored (including the sharp sign itself, obviously).

Comments can be written at any point:

```
# I am a comment using an entire line
DEVICE clock                    # I am a comment after sensor declaration
    DRIVER ClockDriver
        # I am a comment inside a command
        CONFIG
            interval SET 3s
```

Intermediate complexity commands

Following we will cover commands that are rarely used by novices but these commands are needed to be known even by them.

INCLUDE

Syntax:

`INCLUDE {“<URI [*|**]>” | “*”} [ ; ...]`

Purpose:

Insert the content of one or more files (local or remote) into current file. It is really useful for big domotics installations: instead of having one huge file, you divide it into others smaller.

Notes:

While ‘*’ Includes all files in folder, ‘**’ includes also subfolders.

Use of asterisks works only with local files

An URI is one of following (observe that URIs must be between quotes):

- “*”
- file:// (RFC 8089)
- http://
- https://

The first case (”*”) tells the transpiler to look into the *Une* source code for all the occurrences of `DEVICE` command: its associated driver (clause `DRIVER` of `DEVICE` command) will be searched through all Une files inside folder ’include’ ({*home.inc*}) and subfolders.

This is also the action taken by the transpiler when the reserved word `INCLUDE` does not appear in an *Une* source code file. Therefore, the most simple is to copy all needed driver files for a project into {*home.inc*} folder or any subfolder. Obviously, a combination of `INCLUDE` "*" and "file://..." or "http://..." is allowed.

USE

Syntax:

```
USE <name(s)> AS <literal> [; ...]
```

Purpose:

To replace words and symbols. If we take a look to the file “standard-replaces.une”, at certain point, we will find this:

```
USE EQUALS AS ==
```

In other words, the word `EQUALS` is replaced by `==`, which is the *Une* equality operator. To use Spanish language instead of English, we could use:

```
USE IGUAL AS ==
```

When using a declaration like this:

```
USE MyMessage AS “This is a string, so quotes are preserved”
```

The replacements will keep the quotes; but this not the case if left literal has quotes. For instance, following declaration:

```
USE ”MyMessage” AS “This is a string, so quotes are preserved”
```

Is equivalent to the previous one because quotes in left side are ommited.

This command is used to define constants too; which are useful for two reasons:

(a) When a value appears here and there, we can declare it using the `USE` command and we need to change it only in one single place.
(b) We can replace a set (one or more) words for a single one.

This is an example of (a):

```
USE _MAX_TIME_ AS 8s
```

Later we can for instance say:

```
WHEN delay > _MAX_TIME_
```

This is an example of (b):

```
USE NotEquals AS ! equals
```

When using constants inside Strings, you have to surround them using the macro sequence `{*<constant>*}`, as follow:

```
THEN console SET "Delayed for {*_TIME_MAX_*} milliseconds"
```

Notes:

- Replaces can not be nested: you can not refer to a certain replace from another one: if you do it, almost anything can happen (in any universe).
- Time and Temperature units are converted into milliseconds and Celsius respectively after the replacements are done.
- Refer to this file to learn about the standard replaces: <home_install>/includes/standard-replaces.une
- There is a special use case of these 2 keywords (`USE` and `AS`) when using Rules. Please refer to Rules for more information.

- Replacements are done also inside inlined SCRIPTs when the language used in the SCRIPT is the *Une* language.

Advanced commands

Following we will cover commands that are used only by advanced developers or device's manufacturers. But all level *Une* users must have at least a general idea about them.

SCRIPT

Syntax:

```
SCRIPT [<name>]
   LANGUAGE { une |java | javascript | python | ...}
   FROM { "<URI>" [; ...] | {<code>} }
  [CALL "<entry_point>"]
  [ONSTART]
  [ONSTOP]
```

Purpose:

To execute code written in *Une* or other programming languages. They have also another purpose: they are used to create Controllers[27], but this topic is out of the scope of this handbook; it is covered in the book "The Mingle Standard Platform".

Clauses:

- LANGUAGE: The programming language used to create the Script[28][29].

- FROM: can be either a file (compiled or not) or the source code itself embedded in *Une* script code. When embedding the source code, curly-brackets must be used: pay attention to the fact that these green curly-brackets denote that they have to exist.

- CALL: The entry point (when declared) will be invoked after code is properly loaded into the ExEn if the SCRIPT is marked with ONSTART or ONSTOP (also if the SCRIPT is used as a Controller by a DRIVER).

27 Controllers are the last responsible entity for managing physical devices. They are normally created by the manufacturers that fabricate the physical devices.

28 The set of languages that can be used depend on the Mingle Platform version and vendor. The Mingle Standard Platform (MSP) version currently allows (more will be added in the future): Java, JavaScript and Python (notice that when using Java or Python, calls to libraries written in C and C++ can be also used).

29 *Une* is a special language case: only expressions (anything that can be in WHEN and THEN clauses) can be used.

- ONSTART: A SCRIPT marked with this clause will be automatically invoked (its entry point) after ExEn has successfully loaded the model. It will be also be invoked when the script is added (injected) at run-time.
- ONSTOP: A SCRIPT marked with this clause will be automatically invoked (its entry point) just before ExEn will exit. It will be also be invoked when the script is removed at run-time.

Notes:

The SCRIPT command is intended to be used by advanced users, experienced developers and devices manufacturers.

SCRIPT name is optional, but can be omitted only when using either ONSTART or ONSTOP clauses. An error will be reported in all other cases.

DRIVER

Syntax:

```
DRIVER <name>
   SCRIPT <script>
      [CONFIG <name> AS {ANY | <data_type>} [REQUIRED] [; ...]
```

Purpose:

Drivers intermediate between virtual devices and their the physical devices.

Clauses:

- SCRIPT: The name of the script that manages the physical device.
- CONFIG: This clause exposes the configuration parameters that Driver accepts. Modifiers:
  - AS: indicates the type of data expected. ANY means that any *Une* valid data type can be used (we will talk about *Une* data types later).
  - REQUIRED: indicates this parameter must exists (be used in the DEVICE command).

Notes:

End-users do not create DRIVER commands, they just learn about their CONFIG clause to know how to configure them. DRIVER commands are provided by hardware manufacturers.

Please, refer to this file to learn about the drivers that are provided with the Mingle Standard Platform (MSP):

```
<home_install>/includes/common-drivers.une
```

Keywords

These are the words that are used by the *Une* language and therefore they can not be used for any other purpose (like giving a name to a Rule or a Device).

Here are all, alphabetically sorted:

AFTER	ALIAS[30]	ALL	ANY
AS	CALL	CONFIG	DEVICE
DRIVER	FROM	IF	INCLUDE
INIT	LANGUAGE	ONSTART	ONSTOP
REQUIRED	RULE	SCRIPT	THEN
USE	WHEN	WITHIN	

This is a really short set compared with other programming languages.

On the other hand, whenever you find an `INCLUDE` command you should check if referred files contain `USE` commands; if this is the case, you should take them into account.

There is a set of `USE` commands that is recommended to be used because it makes the code more legible. It is recommended but is not needed to be used. This set is declared in a file named “`standard-replaces.une`” and you will find it in all examples that come this the MSP. This file is located in the folder “`include`”.

Data Types

Basic data types

In order to make things as simple as possible, *Une* has only 3 data types:

30 This keyword is not currently in use, but is reserved for future use.

- **Booleans**
  TRUE or FALSE
  Synonyms for TRUE : CLOSED, ON, YES
  Synonyms for FALSE: OPEN, OFF, NO

- **Numbers**
  - A single-precision 32-bit IEEE 754 floating point[31]
  - In the range from: 1.40129846432481707e-45
    to: 3.40282346638528860e+38
  - Decimal separator (when needed) is "." (period)
  - Underscore char ('_') can be used to divide groups of digits (makes easier to read). And you can use as many as you want: they are simply ignored.
  - Binary and Hexadecimal formats are accepted by function `int(...)`, explained later.
  - Examples:
    - `11 or 11.5`
    - `-23.689  or  0.23  or .23`
    - `10_000_000.000_5`

- **Strings**
  Everything in between double quotes: `“This is a string”`

These are known as the basic data types.

Number suffixes

When using numbers, *Une* provides some suffixes to make easier to write time and temperature units.

As time and temperatures are quite common in domotics and as we humans have units for them, *Une* allows to add following suffixes to any number (case is ignored):

Temperature:

- C – for Celsius (the default unit when omitted)

31 Function `int()` allow the use of Binary and Hexadecimal formats.

- F – for Fahrenheit
- K – for Kelvin

Example: 72F → 17º Fahrenheit → 22.2 Celsius

Time:

Managing time consist basically in:

a) Triggering events at a precise moment.

b) Measuring intervals: time elapsed between two moments.

There is an extended explanation about this in the book "The Mingle Standard Platform". In regarding the *Une* language, you need to know that *Une* provides the following suffixes to facilitate writing time constants**:**

- r – for microseconds (1/1000000 seconds)
- l – for millisecond (1/1000 seconds)
- u – for hundred of second (1/100 seconds)
- t – for tenth of second (1/10 seconds)
- s – for second (1 second)
- m – for minutes (60 seconds)
- h – for hours (3600 seconds)
- d – for day (86400 seconds)

Example: *3m → 3 minutes → 180000 milliseconds →* $3*10^9$ <u>*nanoseconds*</u>

Note: you will rarely use units below seconds.

What about Date and Time?

Unlike most known spreadsheets, Dates and Times are not *Une* primitive (basic) data types, but it does not mean that we can not operate with these types of data.

Une allows to manage Dates and Times via functions (spreadsheets do the same). This topic will be covered when explaining the Expression Evaluator (NAJER).

And what about Lists and Pairs?

The same as with Dates and Times: *Une* allows to work with Lists and Pairs via functions, but we will cover them when explaining the Expression Evaluator.

A List is just this, a set of values of any valid *Une* type.

A Pair set is technically named a “dictionary”. It can be considered as a special type of lists: in a dictionary every item of the list is a pair of values (instead of one single value). A dictionary looks like this:

```
name = John Doe,
age = 27,
married = true
```

Technical hint

In *Une*, internally everything is an OOP class, but to make it easier to be used by people without programming skills, *Une* allows to manage basic data (Numbers, Strings and Booleans) as spreadsheets does.

For Date, Time, Lists and Pairs (dictionaries), functions are used, again as spreadsheets does.

Operators

Arithmetic

+	· Sum · Positive number prefix
-	· Subtraction · Negative number prefix
*	Multiplication
/	Division
%	Percent

^	Power	

Relational

==	Equals	Also: IS, EQUALS, ARE
<	Lower than	Also: BELOW
>	Greater than	Also: ABOVE
<=	Lower or equals	Also: MOST
>=	Greater or equals	Also: LEAST
!= or <>	Not equals	Also: UNEQUAL, IS_NOT

Conditional

&&	Logical AND	Also: AND
||	Logical OR	Also: OR
|&	Logical XOR	Also: XOR
!	Logical NOT	Also: NOT

String

+	Concatenate strings
-	Removes from left string all occurrences of right string

Bitwise

&	Bitwise AND	Also: BAND
|	Bitwise OR	Also: BOR
><	Bitwise XOR	Also: BXOR

~	Bitwise NOT	Also: BNOT
>>	Bitwise shift right	
<<	Bitwise shift left	

Others

=	Assignment	Also: SET
:	Send (function invocation)	
,	Used to separate function’s parameters	
\	Code continues to next line (any char after ‘\’ is ignored until next line)	
;	When a clause can have more than one item, this separates one item from another (it can be omitted if each item is written in a new line)	

We will see all these operators in action in a future chapter when covering the Expression Evaluator.

Enunciations

- **Names**: A case-insensitive Unicode-string[32] with following restrictions:
  - A minimum length of one character.
  - A maximum length of 48 characters.
  - Composed by letters and digits in any alphabet and also underscores (“_”) .
  - It can not start with a digit.
  - It can no contain symbols or special characters, neither white spaces.

  Examples: `Name, A_name, añothérNåmè, name23, name_23, _23`

32 In practice it means you can use almost any character of almost any existing language. For more information, visit: https://en.wikipedia.org/wiki/Unicode

- **Literal**: An *Une* valid data type. e.g.: `12`, `true`, `“This is a string”`
- **Expression**: A valid combination of names, properties, literals and operators. Similar to what we place after the “=” in a spreadsheet cell.
- **Value**: Either a literal or an expression that can be resolved to a constant. e.g.:
  - `12 + 3 → 15`
  - `“domotics“ + “ ” + “made easy“ → “domotics made easy”`

<name> or <property>	A combination of letters and digits. Valid chars: [A-Z] [0-9] ‘_‘ (0-9: invalid as first char) Max length: 48 chars
<literal>	Any valid data type: boolean, number or string
<expression>	{<value> | <name>[:<property>]} [<operator> <expression> …]
<value>	<literal> | <expression>* (*) Only if <expression> resolves to <literal>

Chapter 3
Expression Evaluator

Introduction

- An Expression Evaluator is an artifact (a piece of software) that evaluates valid expressions.
- An expression looks like a formula that we put in a spreadsheet cell. More formally: an expression is a valid sequence of: values, names and operators.
- A valid-expression is an expression that can be evaluated by certain Expression Evaluator.
- Different Expression Evaluators can consider the same expression as valid or as invalid.
- Different Expression Evaluators even when considering an expression as valid, can produce different results after evaluating the same expression.
- Not everything is chaos: most Expression Evaluators will accept most of the common expressions and most part of them will produce same result.

Once this said, lets proceed with the Expression Evaluator included in the Mingle Standard Platform (MSP), it is named NAJER, the acronym for: Not Another Java Expressions Resolver.

NAJER follows the same principles as popular expression evaluators such as LibreOffice Calc, MS Excel, Java, Python, JavaScript or C, making it easy to learn if you're familiar with any of these.

It prioritizes simplicity, combining the best aspects of these programs. Slight differences may exist which we will clarify shortly.

One last comment: in the same way you can refer to the value of a spreadsheet cell when writing a formula (e.g. `=C3*2`), when using *Une* “formulas” (we prefer the term “expressions”) you can refer to Devices values by the name of the Device, in this way, you can imagine a Device as a spreadsheet cell.

Technical hint

NAJER is a left-to-right, booleans-lazy expressions evaluator.

Une requires a very unique type of Expressions Evaluator because there can exist parts of the expression that will be resolve in the future (the IF clause of

RULEs).

This makes NAJER unique in its genre, at least as far as we know…

NAJER by example

The best way to show how NAJER works is by examples. We will divide them by the type of the operations. But first, we need to introduce a couple of concepts.

How functions can be invoked

Une allows to invoke a function in two ways:

a) The traditional way, where function parameters are placed after the function name (in between parentheses and comma separated). Function calls are nested.

   Examples (the result is always 4):

   i. floor(4.2)

   ii. min(4, 7)

   iii. min(4, floor(ceiling(7.6)))

   iv. size(left(mid(“0123456789”, 2, 6), 4))

b) The previous way can be hard when you need to nest several functions, like in the last example. So, *Une* provides another way to do the same: using an special sign named “the send operator”, this operator is the semicolon ‘**:**’. You can think about it as sending (invoking) a function using what is at the right of the semicolon as the function parameter.

   Previous examples will be now written as follows:

   i. 4.2:floor()

   ii. 4:min(7)

   iii. 7.6:ceiling():floor():min(4)

   iv. "0123456789":mid(2,6):left(4):len()

There is no difference between one way and the other way, and it is only a question of taste. You can even mix both ways. Although there is an exception: when you use following data types: ‘date’, ‘time’, ‘list’ or ‘pair’, you must use the second one (the send operator).

Technical hint

In the Mingle Standard Platform (MSP), internally everything is an OOP object and the send operator is ‘**:**’. Date, Time, List and Pair are data type classes.

Working with devices

As mentioned, when working with spreadsheets formulas, we can make a reference to another cell, e.g: **`=E3*12`** In a similar way, *Une* expressions can contain references to a declared device’s name.

Lets say we declare a Sensor named “`thermometer_bedroom`” and an Actuator named “`light_kitchen`”. This is how to reference them:

```
WHEN thermometer_bedroom ABOVE 20C \
     OR \
     light_kitchen IS ON
```

Remember: the “\” at the end of a line means that the line continues in next line.

Operators precedence

The same as spreadsheets, Java, Python, JavaScript and others.

Following table shows them sorted from highest precedence to lower precedence.

send	:	e.g.: “Hello world”:mid(2,3)
unary	+ - !	e.g.: +3*2 or -3*2 or !TRUE
power	^	e.g.: 3^2
multiplicative	* / %	

additive	+ -	e.g.: 3+2 or 3-2
relational	> < >= <=	
equality	== !=	
logical AND	&&	
logical OR	||	
logical XOR	|&	
assignment	=	

Arithmetic examples with numbers

12 + 3	15.0	
12 - 3	9.0	
12 * 3	36.0	
12 / 3	4.0	
12 % 200	24.0	
12 ^ 3	1728.0	
0.2 + .3	0.5	.3 is same as 0.3
-12 + (2*4) + 22	18.0	Parentheses are used to group operations and change operators precedence
(-12 + (2*4) + 27) * 3	69.0	
"10" * 2	20.0	Strings representing numbers, when involved in arithmetic operations, are treated as numbers ("+" and "-" operators are special cases)
-8 + "10" * -2	28	

Arithmetic examples with strings

"caco " + "malo"	“caco malo”	“+” concatenate strings
"caco " + "malo" - "o"	“cac mal”	“-” remove substrings
"12" + "34"	“1234”	Concatenate strings
"12" + 34	46.0	Because one operator is a number
"1234" - "3"	“124”	Remove substring
"1234" - 3	1231.0	Because one operator is a number
"12" * "3"	36.0	“*” and “/” are not string operators. Therefore they are always taken as arithmetic operators
"12" / “3”	4.0	
-8 * "10" * -2	-82.0	

Boolean examples

true	true
false	false
true OR false	true
true AND false	false
true AND NOT false	true
8 * 2 EQUALS 16	true
8 * 2 NOT_EQUALS 16	false
NOT (8 * 2 NOT_EQUALS 16)	true
2 < 22	true
(2 < 22) && NOT (8 < 2)	true
(2 < 22) || (4 > 5)	true
(2 < 22) && (4 > 5)	false

(2 < 22) XOR (4 > 5)	true	
"caco" <= "malo"	true	Compared using alphabetical order
"caco" >= "malo"	false	
"caco" < "malo" && (2 < 22)	true	
"caco" > "malo" && (2 < 22)	false	
"caco" == "caco"	true	
"caco" == "CACO"	true	*Une* is case insensitive
“caco":equals("CACO")	false	Use equals() function, to make a sensitive comparison
equals(“caco",“caco",“caco")	true	

Functions examples[33]

Numeric

Min(2, 5, 7, 9)	2.0	
Max(2, 5, 7, 9)	9.0	
floor(3.9)	3.0	
floor(3.7 , 2)	2.0	
FLOOR(-2.5 , -2)	-2.0	No case sensitive
1.58:floor(0.1)	1.5	
ceiling(2.1)	3.0	
ceiling(2.5 , 1)	3.0	
ceiling(-2.5 , -2)	-4	
1.5:ceiling(0.1)	1.5	

33 Later, there is a section about functions syntax and another section about how to invoke them.

round(2.15 , 1)	2.2	
round(-1.475 , 2)	-1.48	
21.5:round(-1)	20	
abs(-3)	3.0	
int(2.8)	2.0	
int("0b11000")	24.0	Binary
int("0x18")	24.0	Hexadecimal
int("1_000_000")	1000000	
mod(10 , 3)	1.0	
rand(0 , 1)	0.0 >= x <= 9.999999	
rand(5 , 50)	5.0 >= x <= 49.999999	
format(“##,###.0”, 123456.789)	“123,456.7”	

String

“1234567”:len()	7 (length)	Alias: size()
char(65) or 65:char()	A	
newLine()	UNIX: "\n" Windows:"\r\n"	OS dependent
“1234567”:reverse()	“7654321”	
“1234567”:left(2)	“12”	
“1234567”:right(2)	“67”	
"A string":lower()	"a string"	
"A string":upper()	"A STRING"	
“john”:proper()	“John”	

"str":search("A string")	3	Index of
"StR":search("A string")	3	No case sensitive
"xyz":search("A string")	0	Not found
search("un","En un lugar de la mancha vivía un...")	4	
search("un","En un lugar de la mancha vivía un...", 7)	33	
search(" ??","En un lugar de la mancha vivía un...")	3	
search(" l*","En un lugar de la mancha vivía un…")	6	
search(" l*","En un lugar de la mancha vivía un…", 9)	15	
"En un lugar de la mancha":mid(7,5)	“lugar”	1st char is 1
"En un lugar de la mancha":mid(13)	“de la mancha”	To the end
"A une passante":mid(7,99)	“passante”	No error
"A une passante":mid(50,99)	“”	No error
"My kingdom for a horse":mid("my","FOR"):trim()	“kingdom”	No case sensitive
mid("12348", 3, 3)	348	Numeric
mid(12348, 3, 3)	348	Numeric
"one,two,three":substitute("o","8")	"8ne,tw8,three"	
"one,two,three":substitute("two","dos")	"one,dos,three"	
"one , two , three":substitute("/s*,/s*", ",")	"one,two,three"	
"one ; two ; three":substitute("/s*;/s*", ";", 2)	"one ; two;three"	Only 2nd is replaced

Miscellaneous

utc() → Current time in milliseconds elapsed since January 1, 1970 UTC	
type(12) or 12:type()	“N” (Number)
type(TRUE) or TRUE:type()	“B” (Boolean)
type(“This is a string”) or “This is a string”:type()	“S” (String)
type(“12”)	“N” (Number)
type(“TRUE”)	“B” (Boolean)
type(date()) or date():type()	“date”
type(time()) or time():type()	“time”
type(list()) or list():type()	“list”
type(pair()) or pair():type()	“pair”
IsReachable(“localhost”)	true
localIPs()	
put(“DoorState”, MyDoorDevice)	“doorstate”
put(“DorrState”, RAND(0,5) > 3)	
get(“DoorState”)	OPEN or CLOSED
del(“DoorState”)	“doorstate”
IIF(time():hour() > 22, 70, 90)	Returns 2nd parameter if expression is evaluated to TRUE and 3rd parameter in other case
iif(time():hour() > 21 && time():hour() < 8, “night”, “day”)	
iif(celsius > 25, "hot", iif(celsius < 17, "cold", "nice"))	
exit()	Finalizes ExEn execution

Functions

In this section we will explain functions syntax in detail: their purpose, their arguments and the types of the arguments.

Numeric functions

Abs

- Description: returns absolute value.
- Syntax: abs(number)
    - *number*: anything that *Une* can interpret as a number. It is required.
- Remarks: An error is launched if parameter can not be interpreted as a number.

Ceiling

- Description: rounds number up, away from zero, to the nearest multiple of significance.
- Syntax: CEILING(number [, significance])
    - *number*: anything that *Une* can interpret as a number. It is required.
    - *significance*: the multiple to which you want to round. When omitted, returns the smallest value that is greater than or equal to its integer
- Remarks:
    - An error is launched if parameter can not be interpreted as a number.
    - Regardless of the sign of number, a value is rounded up when adjusted away from zero. If number is an exact multiple of significance, no rounding occurs.
    - If number is negative, and significance is negative, the value is rounded down, away from zero.
    - If number is negative, and significance is positive, the value is rounded up towards zero.

Floor

- Description: rounds number down, toward zero, to the nearest multiple of significance.

- Syntax: FLOOR(number [, significance])
    - *number*: anything that *Une* can interpret as a number. It is required.
    - *significance*: the multiple to which you want to round. When omitted, returns the largest value that is less than or equal to its integer.
- Remarks: An error is launched if parameter can not be interpreted as a number or if number is positive and significance is negative.

Format

- Description: Formats numbers using the default locale specifications.
- Syntax: format(string, number)
    - *string*: Containing the format to be applied.
    - *number*: to be formatted.
- Remarks: For full description and examples, please refer to following: https://docs.oracle.com/javase/8/docs/api/java/text/DecimalFormat.html

Max

- Description: returns the maximum value from the received ones.
- Syntax: MAX(n1, n2)
    - *n1*: anything that *Une* can interpret as a number. It is required.
    - *n2*: anything that *Une* can interpret as a number. It is required.

Remarks:

- An error is launched if parameters can not be interpreted as a number.
- Float minimum value is returned when zero parameters are passed.

Min

- Description: returns the minimum value from the received ones.
- Syntax: min(n1, n2, ...)
    - *n1*: anything that *Une* can interpret as a number. It is required.
    - *n2*: anything that *Une* can interpret as a number. It is required.
- Remarks:

    - An error is launched if parameters can not be interpreted as a number.
    - Float maximum value is returned when zero parameters are passed.

Mod

- Description: returns the remainder after number is divided by divisor. The result has the same sign as divisor.
- Syntax: mod(number, divisor)
    - *number*: anything that *Une* can interpret as a number. It is required.
    - *divisor*: anything that *Une* can interpret as a number. It is required.
- Remarks: None.

Rand

- Description: generates the next pseudo-random, uniformly distributed value between lower (inclusive) and upper (inclusive).
- Syntax: RAND(lower, upper)
    - *lower*: anything that *Une* can interpret as a number. It is required.
    - *upper*: anything that *Une* can interpret as a number. It is required.
- Remarks: An error is launched if parameters can not be interpreted as numbers.

Round

- Description: rounds a number to a specified number of digits.
- Syntax: round(number [, digits])
    - *number*: anything that *Une* can interpret as a number. It is required.
    - digits: the number of digits to which you want to round the number argument. When omitted, rounds to the closest integer (from 0.1 to 0.4 rounds down and from 0.5 to 0.9 rounds up).
- Remarks:
    - An error is launched if parameter can not be interpreted as a number.
    - If digits is greater than 0 (zero), then number is rounded to the specified number of decimal places.

  - If digits is 0, the number is rounded to the nearest integer.
  - If digits is less than 0, the number is rounded to the left of the decimal point.
  - If digits is not specified, rounds to the closest integer (from 0.1 to 0.4 rounds down and from 0.5 to 0.9 rounds up).

String functions

Char

- Description: Converts the given number into a string representing the Unicode character.
- Syntax: char(number)
  - *number*: to find its corresponding Unicode character. The parameter is required.
- Remarks: For Basic Multilingual Plane (BMP) code points ({@code U+0000}–{@code U+FFFF}), the resulting string will contain exactly one character. For supplementary characters (above {@code U+FFFF}), the resulting string will contain a surrogate pair (length 2).

Equals

- Description: returns true when all received parameters (after being converted into strings) are identical, considering the case.
- Syntax: EQUALS(data1, data2, ...)
  - *dataN*: any valid *Une* data (it will be converted into a string following the rules explained in **Trim()**). At least one parameter is required.
- Remarks: the operator '==' (equality) compare strings ignoring the case. The function EQUALS(...) will consider the case and therefore strings are equals only if they are exactly the same.
  - FALSE is returned when zero parameters are passed.
  - TRUE is returned when only one parameter is passed.

Left

- Description: returns the first character(s) in a string.
- Syntax: LEFT(data, number)

  - *data*: any valid *Une* basic data. It is required.
  - *number*: the number of characters to return. It is required.
- Remarks: while first parameter can be any *Une* basic data type (it will be converted into a string following the rules explained in **Trim()**), the second parameter has to be anything that *Une* can interpret as a number.

Len

- Description: returns the size (in number of characters) of received parameter.
- Syntax: len(data) | size(data)
  - *data*: any valid *Une* basic data. It is required.
- Remarks: if data is not of type string, it will be converted into a string following the rules explained in **Trim()**. An alias for this function is 'size'.

Mid

- Description: Extracts a given specific number of characters from any part of a string. The starting and ending characters can be defined as numbers or as portions of the 'data' string itself.
- Syntax: MID(data, from, to)
  - *data*: any valid *Une* data (it will be converted into a string following the rules explained in **Trim()**). It is required.
  - *from*: if it is a number it will represent the starting index (1 based) of the string. If it is a string, it will be used to search inside 'data' parameter and the first occurrence will be used as the starting index. It is required.
  - *to*: if it is a number it will represent the ending index (1 based) of the string. If it is a string, it will be used to search inside 'data' parameter and the last occurrence will be used as the ending index (this parameter is optional, when to provided, the ending index will be the end of the string).
- Remarks: when the start (from) and/or end (to) points are out of bounds, then an empty string is returned. Start and End has to be anything that *Une* can interpret as a number.

NewLine

- Description: Returns the system-dependent line separator string. On UNIX systems, it returns "\n"; on Microsoft Windows systems it returns "\r\n".

- Syntax: newLine()
- Remarks: There is also a replacement as _NL_.

Right

- Description: returns the last character(s) in a string.
- Syntax: RIGHT(data, number)
    - *data*: any valid *Une* data. It is required.
    - *number*: the number of characters to return. It is required.
- Remarks: while first parameter can be any *Une* basic data type (it will be converted into a string following the rules explained in **Trim()**), the second parameter has to be anything that *Une* can interpret as a number.

Search

- Description: case-insensitive search of a substring in a string.
- Syntax: SEARCH(data1, data2 [, number])
    - *data1*: String to be searched. Any valid *Une* data (it will be converted into a string following the rules explained in **Trim()**). Wildcards '?' and '*' can be used. It is required.
    - *data2*: String to search into. Any valid *Une* data (it will be converted into a string following the rules explained in **Trim()**). It is required.
    - *number*: position (from left) to start the search. It is optional, when not specified, it will be 1: the start of the string.
- Remarks: Returns 0 if the substring was not found or the index (1 based) where the substring starts in the string.

Substitute

- Description: Substitutes *newText* for *oldText* in *source*. When *occurrence* is specified, only nth occurrence is substituted.
- Syntax: SUBSTITUTE(source, oldText, newText [, occurrence])
    - *source*: Text to substitute characters from. Any valid *Une* basic data (it will be converted into a string following the rules explained in **Trim()**). It is required.

  - *oldText*: The text to be replaced (can be a regular expression). It is required.
  - *newText*: The text to replace *oldText* with. It is required.
  - *occurrence*: Specifies which occurrence of *oldText* you want to replace with *newText*. If you specify *occurrence*, only that instance of old_text is replaced. Otherwise, every occurrence of *oldText* in text is changed to *newText* .
- Remarks: None.

Trim

- Description: returns received parameter after removing leading and trailing white chars from beginning and ending from it.
- Syntax: TRIM(data)
  - *data*: any valid *Une* basic data. It is required.
- Remarks: if data is not of type string, it will be converted into a string using following rules:
  - Numbers are converted into their string representation.
  - Boolean TRUE is converted into “`true`” and boolean FALSE is converted into “`false`”.
  - Any other data type (Date, Time, List or Pair) are converted into a string by invoking their “`toString()`” function and then its size is calculated.

Internal cache functions

There is a predefined cache: it works very similar to “pair”s (explained later) and close to `CellSetDriver` (explained in the book “The_Mingle_Standard_Platform”). It is used to store values and access these values later in time.

An example of use:

- `put( “done_on”, date() )`
- `get( “done_on”, “” )`
- `del( “done_on” )`

Put

- Description: saves a value, associates it with a key and returns TRUE.
- Syntax: put(key, value)
    - key: any valid *Une* value.
    - value: any valid *Une* value.
- Remarks: Although any value can be used as 'key', strings are the suggested.

Get

- Description: returns the value previously associated with a key.
- Syntax: get(key [, default])
    - key: any valid *Une* value.
    - default: value to be returned in case the key does not exist (it was not previously set via **Put(…)**). If it is not specified, **""** (empty string) will be used. any valid *Une* value. Can be any valid *Une* value.
- Remarks: None

Del

- Description: deletes an existing 'key,value' pair. Returns TRUE if the 'key' existed and therefore it was deleted, FALSE otherwise.
- Syntax: del(key)
    - key: any valid *Une* value.
- Remarks: If key does not exist, the action is ignored silently (no error is reported).

Miscellaneous functions

Exit

- Description: Finish ExEn returning to the OS the state where ExEn was started.
- Syntax: exit()
- *Remarks: none.*

getTriggeredBy

- Description: returns an instance of “pair” with 2 keys: “name” and “value”, being the value of “name” key the name of the device that triggered the evaluation of the RULE's WHEN and being the value of the key “value” the value of the device that triggered the evaluation of the RULE's WHEN.
- Syntax: *getTriggeredBy*():get(“name”) and *getTriggeredBy*():get(“value”)
- Remarks: *It can only be used inside a RULE.*

IIF

- Description: returns the second parameter if the evaluation of the first parameter is TRUE and the second parameter if it is evaluated to FALSE.
- Syntax: IIF(expression, on_true, on_false)
    - expression: boolean expression to be evaluated.
    - on_true: value to return if expression result is TRUE.
    - on_false: value to return if expression result is FALSE.
- Remarks: This function acts as a ternary operator. It can not be ‘IF’ because it is an *Une* reserved word.

isEmpty

- Description: returns true only if:
    - Received parameter is of type string and it is empty or contains only blank spaces.
    - Received parameter is of type 'list' and it's ::isEmpty() returns true.
    - Received parameter is of type 'pair' and it's ::isEmpty() returns true.
- Syntax: isEmpty(value)
    - value: the vavlue to test.
- Remarks: numeric, boolean, time and date values always return false.

isReachable

- Description: returns true if passed host is reachable with a timeout of passed milliseconds.
- Syntax: isReachable(host, timeout)
  - host: the IP to test.
  - timeout: maximum milliseconds to wait for response from the host.
- Remarks: TCP/IP available is needed.

LocalIPs

- Description: returns an *Une* `list` data type with all known local IPs.
- Syntax: localIPs()
- Remarks: TCP/IP available is needed.

Type

- Description: returns the type of passed argument as explained below under “Remarks”.
- Syntax: TYPE(data)
  - data: any valid *Une* value.
- Remarks: It returns:
  - “N” for numeric
  - “B” for boolean
  - “S” for String
  - “date”, “time”, “list” or “pair” for extended data types.

UTC

- Description: returns the current time in milliseconds elapsed since January 1, 1970 UTC. Note that while the unit of time of the return value is a millisecond, the granularity of the value depends on the underlying operating system and may be larger. For example, many operating systems measure time in units of tens of milliseconds.

- Syntax: UTC()
- Remarks: none.

Extended Data Types

Besides the before mentioned 3 basic data types (Booleans, Numbers and Strings), *Une* also provides another 4 data types:

- Date: to manage dates
- Time: to manage hours, minutes and seconds.
- List: A set of items of any valid *Une* type.
- Pair: A set of {key,value} pairs (a dictionary).

These behave mostly like the basic types and are used in a very similar way, but there are 2 exceptions:

- You have to create them using a function (named constructor).
- When manipulating them, you must use the "send operator" ('**:**')

Lets see this more in detail each one of them.

Date type

Constructor

A 'date' can be created in the following ways:

- String: `date( "2022-06-19" )`
- Numbers: `date( 2022, 06, 19 )`
- Empty: `date()` → today's date

Any other combination of values will produce an error.

Method	Returns	Description

`compareTo(date)`	`number`	Returns 0 if both dates are equals, a positive number if received date is bigger and a negative number if received date is smaller .
`day()`	`number`	Returns the day of the month, from 1 to 31.
`day(number)`	`date`	Changes current date day of month.
`deserialize(str)`	`date`	Creates a date from a previously serialized date.
`duration(date)`	`number`	Returns the amount of days between this date and passed date.
`isLeap()`	`boolean`	Checks if the year is a leap year, according to the ISO proleptic calendar system rules.
`month()`	`number`	Returns the month, from 1 to 12.
`month(number)`	`date`	Changes current date month.
`move(days)`	`date`	Add or subtract days to current date.
`serialize()`	`string`	Creates a string representation of this date. It can be transformed back into same date using the “deserialize” method.
`toString()`	`string`	Returns a string that, when passed to the constructor, creates a date equals to this one. This function is invoked internally when a date is concatenated with the ‘+’ operator.
type()	string	Returns the string “date”.
`weekday()`	`number`	Returns the numeric value for the day of the week. The int value follows the ISO-8601 standard, from 1 (Monday) to 7 (Sunday).
`year()`	`number`	Returns the value for the year.
`year(number)`	`date`	Changes current date year.

Date examples

date()	Today’s date (ISO format)	
date():day() >= 1	true	
date("2021-04-08"):day()	8	1 based
date("2021-04-08"):month()	4	1 based
date("2021-04-08"):year()	2021	
date("2021-04-08"):isLeap()	false	2021 was not
date("2021-04-08"):day(19)	2021-04-19	Set new day
date("2021-04-08"):month(6)	2021-06-08	Set month
date("2021-04-08"):year(2022)	2022-04-08	Set year
date("2021-04-08"):weekday()	4	Monday is 1
date():duration(date("2021-04-14"))	6	days
date():duration(date("2021-04-01"))	-7	days
date():toString()	"2021-04-08"	Today as str
date("2021-04-08") == date("2021-04-08")	true	
date() < date("2999-01-01")	true	
date() > date("2000-01-01")	true	
date("2021-04-08") + 2	2021-04-10	
date("2021-04-08") - 2	2021-04-06	
date(“2021-04-08”):move(2):day()	10	
date(“2021-04-08”):move(-5):day()	3	

Time type

Constructor

A ‘time’ can be created in the following ways:

- String: time("23:45:00")
- Number of elapsed milliseconds since midnight: time(34500)
- 2 numbers: hours, 0 to 23 and minutes, 0 to 59. Seconds is 0.
- 3 numbers: hours, minutes and seconds.

Any other combination of values will produce an error.

Method	**Returns**	**Description**
`compareTo(time)`	number	Returns 0 if both times are equals, a positive number if received time is bigger and a negative number if received time is smaller .
`deserialize(str)`	time	Creates a time from a previously serialized time.
`duration(time)`	number	Returns seconds elapsed between this time and passed time.
`hour()`	number	Gets the hour-of-day field.
`hour(number)`	time	Add or subtract passed hours to current time.
`isDay(lati,long [,date][,zone])`	boolean	Returns true if it is day-time at specified latitude and longitude and optionally at specified day (when date is specified, this instance time is used) and time-zone (when not specified the default one is used).
`isNight (lati,long [,date][,zone])`	boolean	Returns true if it is night-time at specified latitude and longitude and optionally at specified day (when date is specified, this instance time is used) and time-zone (when not specified the default one is used).
`minute()`	number	Gets the minute-of-hour field.
`minute(number)`	time	Add or subtract passed minutes to current time.
`move(number)`	time	Shifts current time passed number of seconds.

`second()`	`number`	Gets the seconds-of-minute field.
`second(number)`	`time`	Add or subtract passed seconds to current time.
`serialize()`	`string`	Creates a string representation of this time. It can be transformed back into same time using the “deserialize” method.
`sincemidnight()`	`number`	Returns number of seconds elapsed since midnight.
`sunNoon (lati,long [,date][,zone])`	`time`	Returns he moment when the Sun crosses the local meridian and reaches its highest position in the sky at specified geoposition; except at the poles (AKA solar noon or high noon). If `’date’` is not specified, today is assumed and time-zone (when not specified the default one is used).
`sunRise (lati,long [,date][,zone])`	`time`	Returns the sunrise at specified geoposition: ‘lat’ latitude and ‘long’ longitude. If `’date’` is not specified, today is assumed and time-zone (when not specified the default one is used).
`sunSet(lati,long [,date][,zone])`	`time`	Returns the sunset at specified geoposition: ‘lat’ latitude and ‘long’ longitude. If `’date’` is not specified, today is assumed and time-zone (when not specified the default one is used).
`toString()`	`string`	Returns a string that, when passed to the constructor, creates a time equals to this one. This function is invoked internally when a time is concatenated with the ‘+’ operator.
type()	`string`	Returns the string “time”.
`twilightSunRise( lat,long [,date] [,zone])`	`time`	Returns the morning twilight at specified geoposition: ‘lat’ latitude and ‘long’ longitude. If `’date’` is not specified, today is assumed and time-zone (when not specified the default one is used).

`twilightSunSet(lat,long [,date] time [,zone])`	Returns the evening twilight at specified geoposition: ‘lat’ latitude and ‘long’ longitude. If ’date’ is not specified, today is assumed and time-zone (when not specified the default one is used).

Time examples[34]

`time()`	`Now (precision = seconds)`	
`time():hour() >= 0`	`true`	`0 >= x <= 23`
`time():minute() <= 59`	`true`	`0 >= x <= 59`
`time():second() <= 59`	`true`	`0 >= x <= 59`
`time(“10:40:10”):hour(3)`	`13:40:10`	
`time(“10:40:10”):minute(22)`	`11:02:10`	
`time(“10:40:10”):second(33)`	`10:40:43`	
`time(“10:40:10”):sinceMidnight()`	`38410`	`Seconds`
`time("10:40:10"):move(-20)`	`10:39:50`	
`time("10:40:10"):move(999)`	`10:56:49`	
`time() == time()`	`true`	
`time("13:10") == time("13:10")`	`true`	`Seconds is 0`
`time("13:10") >  time("13:00")`	`true`	`Seconds is 0`
`time("13:10") <  time("13:20")`	`true`	`Seconds is 0`
`time("13:10") + (60*2)`	`13:12:00`	
`time("13:10") - (60*60)`	`12:10:00`	
`time():sunrise(36.5112,-4.8848,"2021-04-11")(*) == time("07:53:48") → true`		
`time():sunset(36.5112,-4.8848,date("2021-04-11"))(*) == time("20:50:02")`		

34 Time uses 24 hours format: “AM” and “PM” modifiers are not allowed.

```
time():sunset(36.5112,-4.8848,”Europe/Paris”)(*) == time("20:50:02")
```

```
time():sunset(36.5112,-4.8848,"2021-04-11",”Europe/Paris”)(*) ==
time("20:50:02")
```

```
(*) Latitude and Longitude. When no date is passed, today is assumed. When no time-zone is
specified, the default one is assumed.
```

Technical hint

`List` and `Pair` are both ‘iterables’ and provide the usual: `map()`, `reduce()` and `filter()` functions.

These functions work almost exactly as they do in other languages like Java or Python.

List type

A list is just a set of items of any *Une* valid type.

- Lists are 1 based: first item ordinal position is 1.
- The adding order is preserved unless function “sort” is invoked.

Constructor

A ‘list’ can be created in the following ways:

- To create an empty list: list()
- Using an arbitrary number of values: list("A string", 2022, true)
- Using split()[35]: list():split("A string,2022,true”)
- Using split() and separator: list():split("A string|2022|true”, “|”)

Separator is used as being a regular expression, unless it is one single character: in this case it will be properly escaped.

35 Split(...) method uses comma (“,”) as default separator.

Method	**Returns**	**Description**
`add(item,[index])`	`list`	Inserts passed 'item' at the ordinal position of 'index' shifting right all items after 'index'. Negative index counts back from the end of the list, so -1 is the last item of the list, -2 is the last before the last item and so on. To insert before current 1st item, use index 1, to insert before current 2nd item, use index 2, and so on. If ‘index’ is not specified, 'list' is added at the end (tail) of the list. If received parameter is instance of this class (list), the list itself will be appended as one single item.
`addAll ([index,]list)`	`list`	Inserts all items in passed 'list' list at the ordinal position of 'index' shifting right all items after 'index'. Negative index counts back from the end of the list, so -1 is the last item of the list, -2 is the last before the last item and so on. If ‘index’ is not specified, 'list' is added at the end (tail) of the list.
`clone()`	`list`	Returns a swallow copy of itself.
`compareTo(list)`	`number`	Returns 0 if both lists are equals, a positive number if received list is bigger and a negative number if received list is smaller . Comparison is made item by item: using numerical order for these items that are numbers, alphabetical order for string items and ‘true’ is before ‘false’ for boolean items.
`del(value)`	`list`	Deletes 'item' from the list. 1. If received 'value' is a number, it is converted into an integer and the item at the ordinal position of passed 'value' is deleted. 2. If received 'value' is a string and the string

		represents a number, it is converted into an integer and the item at the ordinal position of passed 'value' is deleted. 3. If none of above, 'value' will be searched into the list and if it exist, it will be deleted. 4. If none of the above, an error is thrown. In all cases: • All items after the deleted one will be shifted left. • Negative index counts back from the end of the list, so -1 is the last item of the list, -2 is the last before the last item and so on. • An error is thrown if the index is out of range (or, as said, if the item did not exist in the list).
`deserialize(str)`	`list`	Creates a list from a previously serialized list.
`empty()`	`list`	Deletes all items.
`isEmpty()`	`boolean`	Returns true if this list has no elements. It is equivalent to: len() == 0
`filter(xpr)`	`list`	The expression passed must be a string and must return a `boolean` value, which determines whether or not the processed element should belong to the resulting list. Receives current item's value associated with variable "x" (see example).
`fromJSON(str)`	`list`	Receives an string that represents a JSON Array and creates a list. The Array can contain other JSON arrays and JSON objects (which will be converted into 'pair's).
`get(index)`	`any`	Returns the item at the ordinal position of passed index (1 based: first item is 1).

		Negative index counts back from the end of the list, so -1 is the last item of the list, -2 is the last before the last item and so on.
`has(item)`	`boolean`	Returns true if the list contains passed item.
`index(item)`	`number`	Returns the index of the first occurrence of the specified element in this list, or 0 if this list does not contain the element.
`intersect(list)`	`list`	Deletes from this list all items that does not exist in passed 'list'.
`last()`	`any`	Returns the last item in the list. Produces an error if the list is empty.
`last(default)`	`any`	Returns the last item in the list or ‘default’ if list is empty.
`len()`	`number`	Returns the size (number of items) of the list. ‘size()’ can also be used.
`map(xpr)`	`list`	Takes an expression and returns a new list by applying the expression to each item in the initial list. This expression receives the value of every item associated with the variable "x" (see example).
`peek()`	`any`	Looks at the object at the top of this stack (last added item) without removing it from the stack.
`pop()`	`any`	Removes the object at the top of this stack and returns that object as the value of this function.
`push(item)`	`list`	Pushes an item onto the top (as first element) of this stack.
`reduce(xpr)`	`any`	Allows to compute a result using all the elements present in the list. It receives the partial resulting value and current item value ‘x’,’y’) (see example).

`reverse()`	`list`	Reverses the order of the elements in the list, according to the natural ordering of its elements.
`serialize()`	`string`	Creates a string representation of this list. It can be transformed back into same list using the “deserialize” method.
`set(item,index)`	`list`	Replaces the item at the 'index' ordinal position (1 based: first item is 1) by the passed 'item'. Negative index counts back from the end of the list, so -1 is the last item of the list, -2 is the last before the last item and so on.
`size(maxItems)`	`list`	Sets maximum list length. When adding a new items to the list beyond this limit, all needed items will be deleted. The new items will be added until the maximum size will be reached, ignoring any other remaining items.
`sort()`	`list`	Sorts the list into ascending order, according to the natural ordering of its elements.
`split(str [,sep])`	`list`	Splits the string ‘str’ into its elements using the separator ‘sep’. All elements are added at the end of the list. Note: *‘sep’* can be a regular expression. If not specified, then ‘,’ is used.
`rotate([places])`	any	Rotates the elements in the list. • If no parameter is passed, shifts all elements 1 position to the right. • If 'places' is positive, rotates all elements N positions to the right. • If 'places' is negative, rotates all elements N positions to the left.
`toString()`	`string`	Returns a string that, when passed to the constructor, creates a list equals to this one.

		This function is invoked internally when a list is concatenated with the '+' operator.
`type()`	`string`	Returns the string "list".
`union(list)`	`list`	This is a synonym for: `addAll( list )`
`uniquefy()`	`list`	Remove duplicate items.

Lists examples

`list().len()  (Alias: size())`	`0`	`Length`
`list():get(1)`	`Error out of bounds`	
`list(2.5,”hello”):get(0)`	`Error out of bounds`	
`list(2.5,”hello”):get(1)`	`2.0`	1st `item is 1`
`list(2,5,”hello”):get(2)`	`5.0`	`Length is 3`
`list(2.5,”hello”):get(-1)`	`“hello”`	`Last item`
`list(true,4.2):get(1)`	`true`	`Length is 2`
`list(true,4.2):get(2) == 4.2`	`true`	
`list(true,4.2):add(3.5):last()`	`3.5`	`Length is 3`
`list(true,4.2):add("3.5"):last()`	`3.5`	`Length is 3`
`list():last()`		`Error`
`list():last(-1)`	`-1`	`Default value`
`list(true,4.2):add("hello",1):get(1)`	`“hello”`	`Length is 3`
`list(true,4.2):add("hello"):len()`	`3`	`Alias: size()`
`list(true,4.2):add("hello"):empty():size()`	`0`	`Alias: len()`
`list(true,4.2):del(1).len()`	`1`	
`list(true,4.2):set(”hello”,1):get(1)`	`“hello”`	`Length is 2`

`list():add(3.5):get(1) == list():add(3.5):last()`	`true`	
`list(true,4.2):index(4.2)`	`2`	`Index of`
`list(4,2,1,3):sort() == list(1,2,3,4)`	`true`	
`list(1,2,3):reverse() == list(3,2,1)`	`true`	
`list():split(“1,2,3”):len()`	`3`	`Separa. is “,”`
`list():split(“1|2|3”, “|”):size()`	`3`	`Separa. is “|”`
`list():split(“ one, two, three “, "\s*,\s*")`	`Separator is a RegEx`	
`list(1,2,3):split(“4:5:6”, “:”):get(4)`	`4`	`Length is 6`
`list():split(“1,two,three,4”):get(2)`	`“two”`	`As string`
`list():split(“1,true,false,Hello,Bye”):get(2)`	`true`	`As boolean`
`list(1,2,2,3,4,4):unique() == list(1,2,3,4)`	`true`	
`list(1,2,3,4):map( “x*2” ) == list(2,4,5,6)`	`true`	
`list(1,2,3,4):filter( “x > 2” ) == list(3,4)`	`true`	
`list(1,2,3,4):reduce( “x+y” )`	`10`	
`list(1,2,3,4):filter( “x > 2” ):reduce( “x+y” )`	`7`	
`list(1,2,3,4):filter( “x > 2” ):len()`	`2`	
`list(1,2,3,4):rotate() == list(4,1,2,3)`	`true`	
`list(1,2,3,4):rotate(3) == list(2,3,4,1)`	`true`	

Technical hint

As seen, `List` has 3 functions makes it easier to handle the list as a LIFO stack: `peek()`, `push()` and `pop()`.

Pair type

In computer sciences, a dictionary is a list of pairs. The left element is named “key” and the right element is named “value”. So, a dictionary is a list of keys and values. Keys and values can be of any *Une* valid type. For instance: `name=peter, age=34, married=true`

While values can be any *Use* valid value (any data type), keys must be a basic *Une* data type: String, Number or Boolean. An error will be reported if any other value is used as entry key.

Constructor

A ‘pair’ can be created in the following ways:

- To create an empty dictionary: pair()
- Using an arbitrary even number of values (mod(n, 2) == 0):

  pair("name", “John”, “married”, true, “age”, 32, “date”, date())

- Using split()[36]:

  pair():split("name=John, married=true, age=32”)[37]

- Using split() and declaring the separators:

  pair():split("name:John|married: true|age: 32”, “|”, “:”)

- Separators are used as being regular expressions, unless they are one single character: in this case it will be properly escaped.

Note: Pair makes no guarantees as to the order of the map; it does not guarantee that the order will remain constant after every operation.

Method	**Returns**	**Description**
`add(key,value)`	`pair`	Adds a new pair to the current set of pairs; or replaces current value for a new one if key already existed.
`addAll`(pair)	pair	Add all items in passed dictionary ‘pair’ to this

36 Split(...) method uses comma (“,”) as default separator for entries and equals (‘=’) as separator for key and value.

37 When using split(...), expressions are not allowed; v.g.: date()

		dictionary associated with passed ‘key’.
`clone()`	`pair`	Returns a swallow copy of itself.
`compareTo(pair)`	`number`	Returns 0 if both `pairs` are equals, a positive number if received `pair` is bigger and a negative number if received `pair` is smaller . Comparison is made by comparing the values associated with same keys in both dictionaries; one by one: using numerical order for these values that are numbers, alphabetical order for string values and true is before false for boolean values.
`del(key)`	`pair`	Deletes the 'pair' which key is passed 'key'.
`deserialize(str)`	`pair`	Creates a pair from a previously serialized pair.
`empty()`	`pair`	Deletes all pairs.
`isEmpty()`	`boolean`	Returns true if this dictionary has no elements. It is equivalent to: len() == 0
`filter(xpr)`	`map`	The expression passed must be a string and must return a boolean value, which determines whether or not the processed entry should belong to the resulting map. Receives current entry key’s value associated with variable “x” and current value’s value associated with variable “y”.
`fromJSON(str)`	`pair`	Receives an string that represents a JSON Object and creates a pair. The Object can contain other JSON objects and JSON arrays (which will be converted into ‘list’s).
`get(key[,def])`	`any`	Returns the ‘value’ associated with passed 'key'. If key does not exists and ‘def’ was provided, this value is returned; if it was not provided, “” is returned.

`hasKey(value)`	`boolean`	Returns true if the pair maps a key to the specified value.
`hasValue(value)`	`boolean`	Returns true if the pair maps a value to the specified value.
`keys()`	`list`	Returns a list with all keys of this dictionary.
`map(xpr)`	`map`	Takes an expression and returns a new map by applying the expression to each entry in the initial list. Receives current entry key’s value associated with variable “x” and current value’s value associated with variable “y”.
`reduce(xpr)`	`any`	Allows to compute a result using all the pair’s entries values present in the map (keys are not taken in consideration). It receives the partial resulting value and current value’s value (x,y).
`serialize()`	`string`	Creates a string representation of this pair. It can be transformed back into same pair using the “deserialize” method.
`len()`	`number`	Returns the size (number of pairs) of this set of pairs. ‘size()’ is equivalent (an alias).
`split(str [,s1,s2])`	`pair`	Splits the string ‘str’ into its pairs (Key and Value) using the separator ‘sep1’ for entries and ‘sep2’ for key/values. All found pairs are added to the dictionary. Note: *‘sep1’* and ‘sep2’ can be a regular expression. If not specified, then ‘,’ is used for ‘sep1‘ and ‘=’ is used for ‘sep2‘.
`toString()`	`string`	Returns a string that, when passed to the constructor, creates a dictionary equals to this one. This function is invoked internally when a pair is

		concatenated with the '+' operator.
type()	`string`	Returns the string "pair".
`values()`	list	Returns a list with all values of this dictionary.

Pairs examples

`pair("power=true, red=220, blue=30, green=94")`	`Creates the list`	
`"power=true, red=220, blue=30, green=94":pair()`	`Creates the list`	
`pair(...):get("blue")`	`30`	
`pair(...):get("power")`	`true`	
`pair(...):len()`	`4`	`Alias: size()`
`pair(...):add("name","francisco")`	`Pair added`	
`pair(...):size()`	`5`	
`pair(...):get("name”)`	`"francisco"`	
`pair(...):del(“name”):len()`	`4`	
`pair(...) == pair(...)`	`true`	
`pair():split("name=John,age=36"):get(“age”)`	`36`	
`pair():split("name:John|age:6",”|”,”:”):get(“age”)`	`6`	

AÑADIR MAS

Chapter 4
It is all about Rules

Rule evaluation

Basics

Once again: every time a device changes its state, all rules have a chance to be evaluated. Only for rules that evaluate to true, their **THEN** clause actions will be executed.

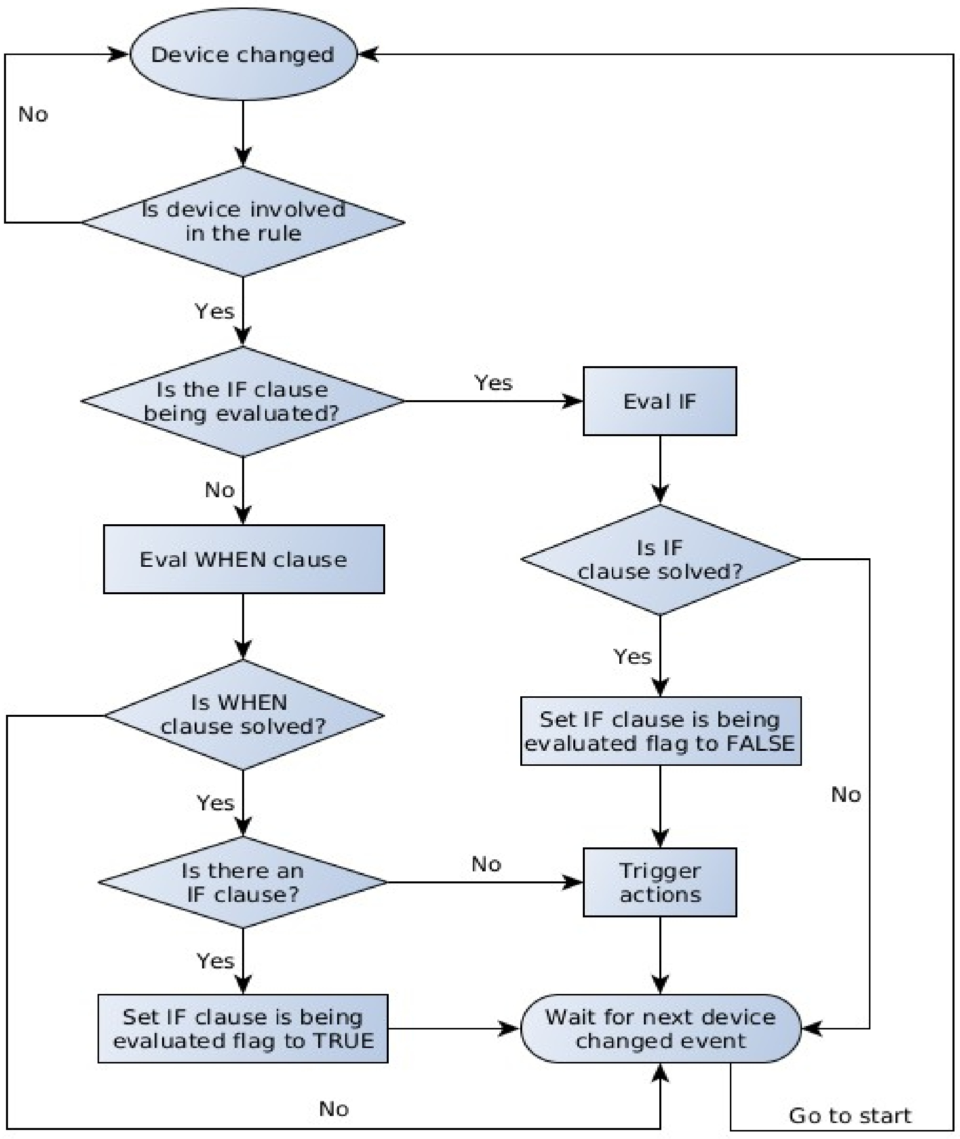
Let us say the same thing in more detail: every time a device (either a Sensor or an Actuator) changes its state (its internal value), all rules are re-visited. Those rules affected by the change (those which **WHEN** clause contain the device that changed) will be re-evaluated; and if rule's **WHEN** clause expression result is true, then all associated actions (rule's **THEN** clause) will be executed[38].

There is another case: when the rule has an **IF** clause. For these rules, when the **WHEN** clause is satisfied, the actions in **THEN** clause will not be automatically executed; prior to it, the **IF** clause must be also satisfied: when both clauses are satisfied (and only if both are satisfied) the actions in **THEN** clause will be executed.

A **RULE** can also be triggered by another **RULE**, but the second **RULE**'s **WHEN** clause will be evaluated before triggering its actions.

The flowing diagram shows how a rule is evaluated (for the sake of comprehensibility, most of the internal details have been concealed):

38 Remember: although all actions will be executed, there is no guarantee about the execution order. Keep this in mind because it can be crucial under some circumstances.


Device changed
No
Is device involved in the rule
Yes
Is the IF clause being evaluated?
Yes
Eval IF
No
Eval WHEN clause
Is IF clause solved?
Yes
Is WHEN clause solved?
Set IF clause is being evaluated flag to FALSE
No
Yes
Is there an IF clause?
No
Trigger actions
Yes
Set IF clause is being evaluated flag to TRUE
Wait for next device changed event
No
Go to start

Triggering sequence

The actions associated with a **RULE**, are triggered in the same order as they appear in the source code. But it does not mean that they are accomplished in the same order as they appear. In other words, although the actions are triggered sequentially, some actions could be accomplished (finish their task) earlier that other actions triggered before.

An action can be one of following actions:

- Evaluate an expression
- Change a (or group of) **DEVICE**'s state

- Execute an **`SCRIPT`**.
- Execute another **`RULE`**.

All actions, except the first one (evaluating an expression), are executed by sending internal messages through the System-Bus. Due to this reason, among others, it is impossible to know the sequence in which the actions will be executed.

However, if an action consists solely of evaluating an expression (e.g. put(“vacations”, true)), it will be executed immediately. Therefore, if all **`RULE`**’s actions are just expressions, they will be executed in the order they appear in the Une source code.

Technical hint

All actions associated with a **`RULE`**, are triggered in same separated thread.

From now and to the end of this handbook, we will use examples to explain most important aspects of rules.

A useless rule

```
WHEN time():hour IS 11
    THEN console SET "This message will never be shown"
```

Explanation: because this rule has no device associated (and rules are evaluated only when an associated device change its value), this rule is useless and an error will be reported by the transpiler.

In order to be executed we can simple add a timer (clock), and replace the time() for the timer’s value (like we’ve seen previously in this handbook).

Without IF clause

Most of the rule’s cases already seen, yield in this category. And as we have seen many of them in previous sections, please refer to those.

With IF clause

Asdfkásñldkfñĺaskdfñĺkasdfñl

Incorrect:

```
IF ((cell == 0 || cell == 9) WITHIN 3s) ...
```

Correct:

```
IF ((cell == 0 WITHIN 3s) || (cell == 9 WITHIN 3s))  ...
```

Triggering a Rule from another Rule

```
RULE rule_1
     WHEN clock BELOW 0
     THEN console SET "Clock value is: " + clock

RULE rule_2
     WHEN clock ABOVE 0
     THEN rule_1
```

"`rule_1`" will never be triggered because its `WHEN` clause will never be satisfied (will never be true). On the other hand, and as explained previously, "`rule_2`" will be triggered every 3 seconds. But now, the action that will be executed when "`rule_2`" is triggered is "`rule_1`", which means that actions declared in "`rule_1`" will be executed: printing in the screen int his example.

So, the result will be the same as the previous example, but in this case, we are using 2 Rules instead of one. Obviously, this has no sense beyond teaching purposes.

The target RULE (the one that is to be triggered) can not have an IF clause.

Considerations when using Date and Time

Following rule will be evaluated whenever a light changes its state:

```
WHEN (ANY light IS ON OR ANY light IS OFF) AND time():hour() IS
19
```

```
    THEN console SET "It is 7PM"
```

But for the purpose of this example, imagine all sensors are push-buttons and all actuators are lights. When a person clicks certain button, certain light is switched on or off (to become the opposite of its previous state).

It could be that no body clicks a button for 60 minutes at 7 PM (or even for days). If this were the case, the previous rule would not be evaluated.

To solve this, we can add a timer and we can associate the timer with the rule: the obvious elapse (and the recommended one) would be 1 hour.

```
DEVICE clock
    DRIVER ClockDriver
        CONFIG
            interval SET 59m

WHEN clock ABOVE 0 AND time():hour() IS 19
    THEN console SET "It is 7PM"
```

Now the rule will be evaluated once per hour. And it will work as expected.

Lets consider another case: we have to check some condition every hour and another condition every 15 minutes, so we could change this timer from 1h to 15m. This is a bad idea!

When doing this, the rule above can be evaluated up to 4 times every hour. It is not a big thing to have 4 messages saying it is 7PM, but there other scenarios were the consequences could be much worse (e.g.: pouring chlorine 4 times in a swimming pool in 1 hour).

So it is better to have 2 clocks: one that ticks every hour and another one every 15 minutes.

All that have been said for time can be applied when working with dates.

Avoiding feedback loops

Feedback occurs when outputs of a system are routed back as inputs as part of a chain of cause-and-effect that forms a circuit or loop. The system can then be said to feed back into itself[39].

This is a well known problem in computer sciences. Lets consider following code:

```
DEVICE clock
    DRIVER ClockDriver
        CONFIG
            Interval SET 1m

DEVICE cell
    DRIVER CellSetDriver
        CONFIG value SET 1

DEVICE console
    DRIVER OutputDriver

WHEN (clock ABOVE 0) AND (cell ABOVE 0) AND (time():hour() IS 22)
    THEN cell    SET clock
         console SET clock
```

In this example, at 22 hours the console will not show just 1 message, but thousands of messages. Lets see why:

1. At 22 hours, the `WHEN` clause is satisfied.
2. Rule's actions (`THEN` clause) will be executed: a request to change the `cell` `ACTUATOR` state will be sent and another one for `console`.
3. When actuators drivers receives the request, they changes the value and send back another message informing about the change.
4. The rule receives both changes (`cell` and `console`). Console changes does not affect the rule because it is not part of its `WHEN` clause. But `cell` is, therefore the rule will be reevaluated.
5. As result of evaluation the `WHEN` clause is satisfied again and we go to 2. This is something that happens in milliseconds or nanoseconds (depending how fast is

39 https://computersciencewiki.org/index.php/Feedback

the hardware were ExEn is running). As result, there will be tons of messages in the console.

Rules must expressed in a way to avoid it.

There are several ways to avoid this.

```
DEVICE clock
    DRIVER ClockDriver
        CONFIG
            interval SET 3m

DEVICE alarm
    DRIVER CellDriver
        CONFIG
            value SET OFF

WHEN alarm IS OFF AND time():hour() IS 22   # This will be true only once
    THEN alarm SET ON
```

To be evaluated faster

To be evaluated less frequently

Lets consider the following situation (we want to):

- Send RAM status to our Telegram every 30 minutes
- Switch ON our house outside light at 22 hours
- Switch OFF our house outside light at 7 hours

- Send a message to the water purifier on Tuesdays to perform an autoclean

We could create a clock that ticks once per hour:

```
DEVICE clock
    DRIVER ClockDriver
        CONFIG
            interval SET 30m

DEVICE RAM
    DRIVER JvmRAMmetricsDriver

DEVICE MyWaterPurifier
    DRIVER DriverForPurifier    # Made up for this example

DEVICE MyTelegram
    DRIVER Telegram
    CONFIG
        chat  SET "ThisIsMyChatId"
        token SET "ThisIsMyTelegramBotToken"

WHEN RAM ABOVE 0                    # This is always true
    THEN console SET "RAM is: " + RAM  + "%"

WHEN clock ABOVE 0 AND time():hour() IS 7
    THEN outside_light SET OFF

WHEN clock ABOVE 0 AND time():hour() IS 7
    THEN outside_light SET OFF

WHEN clock ABOVE 0 AND date():weekday() IS TUESDAY
    THEN MyWaterPurifier SET "autoclean"
```

But doing it, the weekly and monthly `WHEN` condition rules are being evaluated every hour: not very efficient (despite the feedback loop problem explained previously). It is a much better idea to create 3 clocks like this:

```
DEVICE light_clock
    DRIVER ClockDriver
        CONFIG
            Interval SET 1h
```

```
DEVICE purifier_clock
    DRIVER ClockDriver
        CONFIG
            Interval SET 1d

DEVICE report_clock
    DRIVER ClockDriver
        CONFIG
            Interval SET 1d
```

Other rules optimizations

In this section, we will

And that’s all, folks