OFFICIAL DOCUMENTATION

LINE 2.0

// Language Reference Manual

INDEX

01 Comments

In this follows the syntax of languages such as C:

SYNTAX

```
// For single-line comments
/* For multiline comments
```

```
    and to close */
```

02 Variables

A variable has this form:

SYNTAX

```
name_variable = content
```

In the name you can only use letters, numbers (not in first position) and the low hyphen (`_`).
Any character can be used in the content.

Types of Data

There are 3 types of data:

LIKE	DESCRIPTION
`Stringa`	Store data for how you see it.
`Numerale`	He stores the number not so much for how he sees himself, but for his value.
`Booleano`	It has only two possible values: `V`(Verum) and `F`(Falsum).

To indicate that a variable stores a numeral or boolean it is necessary to precede the name of the variable with `$`.

EXAMPLE

```
var = 8 // stores a string (8)
$ var = 8 // stores a number (8)
var = V // stores a string (V)
$ var = V // stores a Boolean (V)
```

> If in a variable preceded by `$` You store something that can be neither numeral nor Boolean, it will result in an error.

Type conversion

From string to numeral — the command is used `n:`

- If the string is a number, the transform will have the equivalent value.
- If the string doesn't just have numbers, it'll be worth `1`.
- If it's an empty string, it'll be worth `0`.

EXAMPLE

```
var = 8
n: var // n: is the command to numericate
```

From string to boolean — you use the command `b:`

- If it is an empty string, it will be converted to `F`.
- Otherwise it will be `V`.

EXAMPLE

```
var = hello
s: var // now var is a Boolean that is worth V
```

From number to boolean and vice versa

- If it is worth zero, it converts to `F`, otherwise it becomes `V`. Conversely, `V` becomes `1` and `F` becomes `0`.

EXAMPLE

```
$ number = 5
$ bool = V
b: number // = V
n: bool // = 1
```

From number/boolean to string — it is used `s:`

The value becomes simply readable for the human, but does not change structure.

EXAMPLE

```
$ number = 5
$ bool = V
s: number // = 5
s: bool // = V
```

> To know the type of a variable, the command is used `\t` (where `\` is the line escape). Names are given in English: `string`, `number`, `bool`.

Reassignment

A new value can always be reassigned to a variable.

EXAMPLE

```
var = 5
var = 10
n: var
```

So we'll have one `var` numeral that stores `10`.

03 TALK — Unified output and input

To interact with the user, the keyword is used `TALK`. This combines output and input

in a single intuitive statement.

Behavior of TALK

`TALK` prints the text provided and, if it meets a variable not yet existing (preceded by `@`), automatically asks the user to enter a value for that variable.

OUTPUTS ONLY (ALL EXISTING VARIABLES)

```
Name = Mario
TALK Hello @ name !
```

OUTPUT

```
Hello Mario!
```

OUTPUT + INPUT (NON-EXISTENT VARIABLE)

```
Name = Mario
TALK Hello @ name @ surname
```

OUTPUT

```
Hello Mario [requires input for 'surname']
\\user inserts: Rossi
Hello Mario Rossi
```

FULL EXAMPLE

```
TALK What's your name? @ name
TALK How old are you? @ eta
TALK Hello @ name, you are @ etaage!
```

OUTPUT

```
What's your name? [requires 'name']
\\user: Marco
How old are you? [requires 'age']
\\ user: 25
Hi Marco, you are 25 years old!
```

Syntactic sugar — OUT and INP

For convenience, LINE also maintains traditional keywords `OUT` and `INP` as shortcuts:

COMMAND	BEHAVIOR	EQUIVALENT TALK
`OUT`	Only output, no input	TALK with all existing variables
`INP`	Only input, no text before	TALK with only non-existent variables

OUT — OUTPUT ONLY

```
var = hello
OUT @ var world!
```

OUTPUT

```
Hello world!
```

INP — INPUT ONLY

```
INP Name: @ first name \nSurname: @ surname
OUT Hello @ name @ surname
```

OUTPUT

```
Name: [input]
Surname: [input]
Hello [name] [surname]
```

Special Characters

> The "power" of `@` ends with any character that is not inclusive in a variable name (`a-z`, `0-9`, `_`).

VARIABLE TYPE WITH \T

```
var = hello
OUT \t var
```

OUTPUT

```
String
```

In this case `@` You can omit because `\t` It only works with existing variable names.

Escape and spaces

Any space except the first one after `TALK` / `OUT` It is considered. If you want to print the snail you can use the escape `\`.

EXAMPLE

```
var1 = Hello
var2 = world!
TALK @ var1 @ var2
TALK @ var1 @ var2
TALK \ @ var1
```

OUTPUT

```
Hello world!
Hello world!
@var1
```

The command `\n` It is to indicate to go to terms.

04 Notes on INP (deprecated)

> **Note:** `INP` is maintained for backward compatibility, but `TALK` is the recommended method for managing input and output in a unified manner.

If you still use `INP`, the behavior is:

INP SYNTAX

```
INP @ var
```

> If you put the name of an existing variable in the input, `INP` will not ask for any input but will simply display the content of the variable.

05 Mathematical operations

Among the numbers, the following mathematical operations can be carried out:

OPERATION	OPERATOR
Addition	`+`
Subtraction	`-`
Multiplication	`*`
Division	`/`
Whole part division	`//`
Rest Division	`%`

> These operations can only be performed within variables. Even in a numeric variable the various signs will be interpreted as operations. Only the result will be stored.

06 Comparative operations

You can take place between any type of data:

OPERATION	OPERATOR
Equality	`==`
Diseguaglianza	`≠`

> Un `5` numerale è diverso da un `5` stringa.

Queste invece si possono eseguire solo fra numeri:

OPERATION	OPERATOR
Major	`>`
Minor	`<`
Minore o uguale	`⩽`
Maggiore o uguale	`⩾`

Queste operazioni non possono essere svolte all'interno di variabili, ma solo all'interno di IF block, e danno come risultato un booleano.

07 Logical operations

Operable only among Booleans:

OPERATION	KEYWORD
AND	`ET`
OR	`VEL`
XOR	`AUT`
NOT	`!` (prefisso)

Operable only in IF block, they give as a result a Boolean. They can also be used to concatenate comparison operations.

EXAMPLE

```
@ var < 5 ET @ var > 6
```

08 IF Block

Un IF block viene definito così:

SYNTAX

```
IF name_if = condition
```

PARTE	DESCRIPTION
`IF`	La parola chiave.

PARTE	DESCRIPTION
`nome_if`	The unique name of the block.
`condizione`	The condition: if true (`V`) you run the GO that recalls this IF.

EXAMPLES

```
IF prova  = 5 == 3            // questo è falso
IF prova2 = 5 ≠ 3             // questo è vero
$var = 5
IF prova3 = @var == 5         // questo è vero
IF prova4 = \tvar == number   // questo è vero
```

09 Functions

Le funzioni sono dei blocchi di codice che descrivono senza eseguire.

SYNTAX

```
FUN nome_funzione(parametri)
  ...
  return
  FEND
```

ELEMENT	DESCRIPTION
`FUN`	Key word.
`nome_funzione`	Nome univoco della funzione.
`parametri`	Valori di variabili esterne da considerare. Se ne possono mettere quanti se ne vogliono (anche nessuno), separati da virgola.
`...`	Codice della funzione.

ELEMENT	DESCRIPTION
`return`	Optional; if included, it must be put to the end. The return value can be saved in a variable or printed. It can only include one per function.
`FEND`	Chiude il blocco.

Chiamata di una funzione

ESECUZIONE

```
nome_funzione()
```

SALVARE IL VALORE DI RITORNO

```
name_variable = name_function()
```

ESEMPIO SEMPLICE

```
FUN ciao()
  OUT ciao
  FEND
ciao()
```

OUTPUT

```
Hello
```

EXAMPLE WITH PARAMETERS AND RETURN

```
Name = Mario
FUN Ciao (initial)
  OUT hello @ initial !
  $ number = 5
  return @ number
  FEND
Hello(@ name)
Val = hello(@ name)
OUT @ val
```

OUTPUT

```
Hi Mario!
5
```

Then blocks

There are special functions that do not accept arguments and do not return values: blocks `THEN`. The structure is similar to that of the functions, but it must be replaced `FUN` with `THEN` and `FEND` with `THEND`, and it is not possible to include a `return`.

SYNTAX THEN

```
Then name_then
  ...
  THEND
```

Which is equivalent to:

EQUIVALENT WITH FUN

```
FUN name_function()
  ...
  FEND
```

> The use of `THEN` It is purely optional and only for convenience. This type can only be called in a GO block.

10 GO Block

To perform a certain function (usually a `THEN`, as the GO does not access functions that accept parameters and return values) in relation to a certain IF, the GO block is used.

SYNTAX

```
GO # name_if # name_function() # repetitions

// or if you have used THEN instead of FUN

GO # name_if # name_then # repetitions
```

ELEMENT	DESCRIPTION
`#nome_if`	Name of the IF block to which it refers.
`#nome_funzione()` / `#nome_then`	Name of the function to which it relates. As already mentioned, the function cannot include `return` and cannot accept arguments; it is good practice to use `THEN` for clarity.
`#ripetizioni`	Times it has to be run in a row. If equal to `1` You can omit. Is there a special value `c`: with it the GO will repeat itself as long as the condition is true.

Examples

SINGLE EXECUTION

```
IF se = V
Then then
  OUT Yay!
  THEND
GO ♪ if #
```

OUTPUT

```
Long live!
```

REPEATS WITH VARIABLE

```
$ count = 2
IF se = V
Then then
  OUT Yay!
  THEND
```

```
GO  if   #@
```

OUTPUT

```
Long live!
Long live!
```

REPEATS WITH FIXED NUMBER

```
IF se = V
Then then
  OUT Yay!
  THEND
GO # if #  #
```

OUTPUT

```
Long live!
Long live!
Long live!
```

REPEATS WITH C (AS LONG AS THE CONDITION IS TRUE)

```
$ count = 0
IF if = @ counts < 3
Then then
  OUT Yay!
  count = @ count + 1
  THEND
GO ♪ if   # c
```

OUTPUT

```
Long live!
Long live!
Long live!
```

Concatenation of conditions – elif / else

If you want that in case you do not meet the initial condition you perform another

action under another condition, after the three parts of the GO you add three more with `&`. It is equivalent to a `elif` of Python.

If you use it `&&`, is interpreted as the `then` to be performed in the event that all the above conditions are false. It is equivalent to a `else` of Python. You can only put it to the end; you can't add more `&` after it. Only one condition at a time can be true.

COMPLETE STRUCTURE ELIF/ELSE

```
GO # if_1 # then_1 # repetitions_1 & # if_2 # then_2 && #  # repetitions_3
```

11 Lists

You can create variables that store multiple values: **lists**.

SYNTAX

```
list = [hello ; salve]
```

The list must be included in square brackets. Each element is separated from `;` (use `\` to indicate that `;` is an element of the list and not a divider).

Indices and tags

Each element is associated with:

- An **index**, which indicates the position in the list and starts from `0`. For example `ciao` has index `0` and `salve` has index `1`.
- One or more **tags**, which can be added with `|`. If you want to use `|` as a character use the escape `\`.

LIST WITH TAGS

```
list = [greeting | hello]
```

LIST WITH MULTIPLE TAGS

```
list = [Italian | greeting | |hello ; salve]
```

In this case `ciao` has two tags (`saluto` and `italiano`) and `salve` no one.

Lists nested

Elements of a list can be lists themselves:

LIST NESTED

```
list = [names | [Giulio ; Carlo] ; surnames | [Whites ;; Ferrari]]
```

Add elements — ADD

SYNTAX

```
ADD name_list AT index = value
```

ELEMENT	DESCRIPTION
`ADD`	Key word.
`nome_lista`	Name of the list to add to.
`AT`	Key word. The index must be less than or equal to the last index; the one that was already in that index will be moved one forward. If left empty, it goes to the last place.
`valore`	Value to add.

INSERT IN SPECIFIC POSITION

```
list = [hello ; salve]
```

```
ADD list AT 1 = good morning
OUT @ list
```

OUTPUT

```
[hello ; good morning ; salve]
```

INSERTION AT THE BOTTOM

```
list = [hello ; salve]
ADD list AT = good morning
OUT @ list
```

OUTPUT

```
[hello ; salve ; good morning]
```

To **replace** an element instead of `AT` is used `IN`:

REPLACEMENT WITH IN

```
list = [hello ; salve]
ADD list IN 1 = good morning
OUT @ list
```

OUTPUT

```
[hello ; good morning]
```

Delete items – CANC

SYNTAX

```
CANC name_list AT index // by index
CANC nome_listaname_list IN tag // by tag
CANCE name_list IS value // by value
```

> If two items have a tag in common, both will be deleted. You can also choose a list of tags:

```
CANC nome_lista IN [tag1 ; tag2]
```

Recalling elements

To recall the lists we use `@` (Don't give lists and variables the same name, they're the same thing!).

ACCESS BY INDEX/TAG

```
name_list[index]
name_list[tag]
name_list[tag1 ; tag2]
```

A list's values are just strings, but you can always convert when you call back. If you print a list, only the values will be displayed and not the tags. To view the tags you also use `tOUT` in place of `OUT`.

OUT VS TOUT

```
lista = [saluto | ciao]
OUT  @lista
tOUT @lista
```

OUTPUT

```
[greeting]
[greeting | hello]
```

12 Other Keywords

Variables – RET / DEFRET / TEMP

By default, a variable in a block is only accessible within the block itself or internal

blocks. To make a global variable it puts it forward `RET` to the name.

If, on the other hand, you want all variables to be global in a given block, you use the word at the beginning of the block `DEFRET`. After that, if you want a single variable to remain non-global, it prefixes `TEMP`.

EXAMPLE

```
FUN ciao()
  RET $ one = 1
  $$three = 3
  FEND
FUN cioa()
  DEFRET
  $ two = 2
  TEMP $$four = 4
```

In this case the variables `tre` and `quattro` are not accessible from the outside; instead `uno` and `due` Yes.

Operator IN

`IN` is used to determine whether an item is in a given set of values (e.g. a list). If it is present, the result is `V`, otherwise it is `F`.

EXAMPLE

```
list = [hello]
IF exists = hello IN list
Then press
  OUT is there!
  THEND
GO # exists # press
```

OUTPUT

```
There is!
```

Constants — STAY

`STAY` is put before the name of a variable to make it unchangeable (constant).

EXAMPLE

```
STAY hello = hello
Hello = that is // error!
```

13 Error management

Error codes

LINE ERROR CODES	
VAR_NOT_FOUND	Non-existent variable
TYPE_ERROR	Invalid data type
INVALID_CONVERSION	Type conversion not possible
CONST_MODIFY	Attempt to modify a constant (STAY)
DIV_BY_ZERO	Division by zero
MATH_ERROR	Invalid mathematical operation
BOOL_EXPECTED	Expected Boolean value
NUMBER_EXPECTED	Expected numerical value
STRING_EXPECTED	Waiting string
LIST_NOT_FOUND	Non-existent list
LIST_OUT_OF_RANGE	Index outside the limits of the list
LIST_EMPTY	Empty list
TAG_NOT_FOUND	Non-existent tags in the list
DUPLICATE_TAG	Duplicate tag not allowed
FUNC_NOT_FOUND	Non-existent function
INVALID_ARGUMENTS	Invalid topics
RETURN_NOT_ALLOWED	Use of return not allowed
THEN_ARGUMENTS	Use of topics in a THEN block
IF_NOT_FOUND	Non-existent IF block
GO_INVALID	Invalid GO block
GO_LOOP_ERROR	Infinite or invalid loop
INPUT_ERROR	Error in input
OUTPUT_ERROR	Output Error

SYNTAX_ERROR	Syntax error
ESCAPE_ERROR	Invalid escape
UNKNOWN_ERROR	Unknown error
MOD_NOT_FOUND	Form not found

TRY / SHOW / YET

`TRY`, `SHOW` and `YET` allow you to manage execution errors without abruptly interrupting the program. `SHOW` and `YET` They are optional.

STRUCTURE

```
TRY name_try
  ...
  TREND

SHOW name_show # name_try
  ...
  SEND

YET # name_try
  ...
  YEND
```

EXAMPLE TRY / SHOW

```
TRY test
  $ a = 10 / 0
  TREND

SHOW manage # try
  OUT Error
  SEND
```

Error variables

When an error is made managed by LINE, in the `SHOW` The variables are available:

VARIABLE	LIKE	DESCRIPTION
`ERR_CODE`	Numerale	Se è un errore gestito da LINE (vedi la lista sopra), vale `1`.

VARIABLE	LIKE	DESCRIPTION
`ERR_MSG`	Stringa	Descrizione testuale dell'errore.

> Le variabili `ERR_CODE` e `ERR_MSG` esistono solo all'interno dello SHOW.

ESEMPIO CON ERR_CODE / ERR_MSG

```
TRY prova
  OUT @inesistente
  TREND

SHOW gestisci #prova
  OUT @ERR_CODE
  OUT @ERR_MSG
  SEND
```

YET — guaranteed execution

The block `YET` is always executed, both in case of error and in case of correct execution.

ESEMPIO YET

```
TRY prova
  $a = 10 / 2
  TREND

YET #prova
  OUT Fine operazione
  YEND
```

Comportamento di default

Di default LINE assume questo comportamento: se un errore avviene all'interno di un `TRY` e non è intercettato da nessuno `SHOW`, allora:

- Assegna `ERR_CODE = 1`
- Print `ERR_MSG`
- End the execution of the program

> Important rules:
>
> - `TRY` cannot be nested
> - Every `TRY` can have only one `SHOW`
> - Every `TRY` can have only one `YET`
> - `ERR_CODE` ed `ERR_MSG` exist only in the `SHOW`

14 Modules

The modules are files with code within them, divided into functions. To use these functions within your file you must first import the module; the functions inside it refer to normal functions.

A function in a module has a compound name as follows: `nomemodulo_nomefunzione()`.

SYNTAX IMPORT

```
TAKE namemodulo
```

EXAMPLE

```
TAKE math
var = math_sqrt(9)
OUT @ var
```

OUTPUT

```
3
```

> To know the list of forms, type `.MODS` in a blank file and run the code: in the console you will have the full list of modules.

> To see the list of functions of a module and their operation, type the uppercase name of the form preceded by a point (e.g. `.MATH`) in an empty file and run.

15

File LINE

A file with LINE code is saved with extension `.line`.

FILE EXTENSION

```
namefile.line.line
```