Manuale LINE 3.0

Language with Intuitive and Natural Expression

Indice

1 Introduzione

1.1 Cos'è LINE?

LINE è un linguaggio di programmazione progettato per essere comprensibile e facile da apprendere. La sintassi mette al centro la logica, rendendola leggibile quasi come testo naturale.

1.2 Cosa significa LINE?

LINE è un acronimo: **Language with Intuitive and Natural Expression**.

Un programma LINE viene salvato in un file con estensione `.line` ed eseguito dall'interprete.

2 Commenti

```
>> Commento su singola riga

<* Commento
   multilinea *>
```

3 Escape

Il carattere di escape è il backslash \. Precede caratteri speciali per dargli un significato diverso da quello letterale.

Sequenza	Significato
`\n`	Vai a capo (newline)
`\\`	Backslash letterale
`\@`	`@` letterale (non riferimento a variabile)
`\%`	`%` letterale (`%` senza escape è invisibile in `TALK`)
`\;`	`;` letterale in una lista (non separatore di elementi)
`\\|`	`|` letterale in una lista (non separatore di tag)
`\tNOME`	Tipo della variabile `NOME` (restituisce `string`, `number` o `bool`)

Nota su `%`: il simbolo `%` scritto direttamente in un `TALK` è invisibile nell'output. Se contenuto in una variabile, viene stampato normalmente. Per stamparlo direttamente usa `\%`.

4 Variabili

Una variabile ha questa forma:

```
TIPO_VARIABILE nome_variabile = dati
```

- `TIPO_VARIABILE` — indica la tipologia di variabile (vedi sezione 6). Può essere omesso per le variabili standard.

- `nome_variabile` — nome univoco. Può contenere solo lettere, numeri (non in prima posizione) e il trattino basso `_`.
- `dati` — il valore immagazzinato. Se c'è più di un dato si tratta di una lista (vedi sezione 15).

Per riferirsi al valore di una variabile da un altro punto del codice si usa `@` davanti al nome:

```
nome = Mario
TALK Ciao, @nome!
<* Output: Ciao, Mario! *>
```

Il "potere" di `@` si interrompe al primo carattere non ammesso nel nome (spazio, `!`, `,`, ecc.).

5 Tipi di dato

Esistono tre tipi di dato:

Tipo	**Prefisso**	**Descrizione**
Stringa	*(nessuno)*	Immagazzina il dato così com'è, come testo
Numerale	`#`	Immagazzina un valore numerico
Booleano	`#`	Solo due valori: `V` (vero) o `F` (falso)

Il prefisso `#` davanti al nome dichiara che la variabile contiene un numerale o un booleano:

```
var = 8      >> stringa: immagazzina il testo "8"
#var = 8     >> numerale: immagazzina il numero 8
var = V      >> stringa: immagazzina il testo "V"
#var = V     >> booleano: immagazzina il valore vero
```

Se in una variabile con `#` si immagazzina qualcosa che non può essere né numero né booleano, si ottiene un errore di tipo.

5.1 Conversioni di tipo

Comando	**Effetto**
`n:var`	Converte in numerale. Se la stringa è un numero usa quel valore, altrimenti vale `1`. Stringa vuota → `0`.
`b:var`	Converte in booleano. Stringa vuota o `"F"` → `F`, tutto il resto → `V`. Da numero: `0` → `F`, qualsiasi altro → `V`.
`s:var`	Converte in stringa. Il valore diventa testo leggibile.

```
var = 8
n:var        >> ora var è il numerale 8

#n = 0
b:n          >> ora n è il booleano F

#n = 5
```

```
s:n          >> ora n è la stringa "5"
```

Per sapere il tipo attuale di una variabile si usa `\t` seguito dal nome:

```
#eta = 25
TALK \teta   >> Output: number

nome = Mario
TALK \tnome  >> Output: string
```

6 Tipi di variabile

Esistono tre tipi di variabile.

6.1 Standard *(nessun prefisso)*

La variabile più comune. Il valore può essere cambiato con una **riassegnazione** — stessa sintassi della dichiarazione:

```
var = Buongiorno   >> dichiarazione
var = Arrivederci  >> riassegnazione
```

Si può anche cambiare tipo durante la riassegnazione:

```
var = Ciao    >> stringa
#var = V      >> riassegnata come booleano
```

> **Importante:** se una variabile viene posta uguale a un'altra e quella cambia, la prima **non cambierà** — il valore viene copiato al momento dell'assegnazione.

```
#a = 5
var = @a    >> var vale 5 (copia di a)
#a = 6      >> a cambia, ma var rimane 5
```

6.2 STAY — costante

Si antepone `STAY` alla dichiarazione. Il valore non può essere cambiato in nessun modo.

```
STAY #pi = 3.14
STAY lingua = italiano
```

6.3 IF — reattiva

Si antepone `IF`. Queste variabili possono contenere solo booleani o operazioni che producono booleani. Il prefisso `#` non è necessario.

```
IF vero = V          >> corretto
IF errore = 5        >> ERRORE: 5 non è un booleano
```

La caratteristica che le rende speciali: se una variabile IF è posta uguale a un'altra variabile e quella cambia, **anche la IF cambia di conseguenza**.

```
#a = V
IF cond = @a
#a = F    >> adesso anche cond vale F
```

7 Console

La console è lo spazio dove il programma interagisce con l'utente.

7.1 TALK

`TALK` è la parola chiave principale. Tutto ciò che segue viene mostrato nell'output.

```
ciao = ciao
TALK @ciao mondo!
<* Output: ciao mondo! *>
```

Se `@` fa riferimento a una variabile **inesistente**, la console chiede automaticamente un input e lo salva in quella variabile:

```
TALK Ciao, inserisci i tuoi dati \nNome: @nome \nCognome: @cognome
TALK Ciao @nome @cognome
```

7.2 OUT e INP

Varianti di `TALK` considerate zucchero sintattico:

Parola chiave	Comportamento
`OUT`	Solo output. Con variabili inesistenti dà errore invece di chiedere input.
`INP`	Richiede almeno un input.

Usa sempre `TALK`. `OUT` e `INP` possono risultare utili per chiarezza in certi contesti.

7.3 SCREAM

Stampa una lista mostrando anche i tag (al contrario di `TALK` che mostra solo i valori):

```
lista = [saluto | ciao]
TALK @lista    >> [ciao]
SCREAM @lista  >> [saluto | ciao]
```

7.4 TALK:V e TALK:F — input persistente

`TALK:V` attiva un input persistente che appare sempre a fine output in console. L'input inserito viene salvato nella variabile speciale `ASKED`.

```
TALK:V Inserisci un comando:
TALK Hai scritto: @ASKED

TALK:V Nuova domanda:   >> sovrascrive la domanda precedente
```

```
TALK:F                  >> disattiva l'input persistente

TALK:V                  >> riattiva senza domanda
```

> **Attenzione:** se si attiva un input con `TALK` o `INP`, oppure nessuna parte del codice richiama `ASKED`, `TALK:V` si disattiverà automaticamente.

7.5 Formattazione testo console

È possibile colorare e stilizzare il testo nella console usando la sequenza di escape `\C{}`:

```
TALK \C{red+bold+_}Attenzione!\C{red}: \C{}salutare!
```

Sintassi: `\C{colore+modificatori}`

I modificatori si combinano con `+`. Per ripristinare il colore di default si usa `\C{}` con le graffe vuote. Lo stile si applica a tutto il testo successivo fino al prossimo `\C{}`.

Modificatore	Effetto
`bold`	Grassetto
`italic`	Corsivo
`-`	Barrato
`_`	Sottolineato

Colori nominali disponibili: `red`, `green`, `blue`, `yellow`, `orange`, `pink`, `cyan`, `purple`, `teal`, `white`, `gray`, `amber`, `lime`, `navy`, `gold`, `magenta`, `brown`, `sky`.

È possibile usare anche colori esadecimali: `\C{#ff9100}`.

```
TALK \C{red+bold+_}Attenzione!\C{red}: \C{}testo normale
<* "Attenzione!" in grassetto rosso sottolineato,
   ":" in rosso, resto normale *>

TALK \C{#00e5ff}Testo ciano esadecimale\C{}
```

8 Operazioni matematiche

Le operazioni matematiche si svolgono solo tra variabili numeriche, all'interno di una dichiarazione o riassegnazione. Viene immagazzinato solo il risultato.

Operatore	Operazione
`+`	Addizione
`-`	Sottrazione
`*`	Moltiplicazione
`^`	Potenza
`/`	Divisione
`//`	Divisione intera (parte intera del risultato)
`%`	Resto della divisione (modulo)

Il simbolo `+` funziona anche tra stringhe come concatenazione.

9 Operazioni di comparazione

Producono sempre un booleano (`V` o `F`). Possono essere usate all'interno di qualsiasi variabile.

Operatore	Significato	Tra
`==`	Uguale	Qualsiasi tipo
`!=`	Diverso	Qualsiasi tipo
`>`	Maggiore	Solo numeri
`<`	Minore	Solo numeri
`>=`	Maggiore o uguale	Solo numeri
`<=`	Minore o uguale	Solo numeri

Un `5` numerale è diverso da un `5` stringa — il tipo conta.

```
#var = 5
IF uguale   = @var == 5    >> V
IF diverso  = @var != 3    >> V
IF maggiore = @var > 3     >> V
IF tipo     = \tvar == number  >> V
```

10 Operazioni logiche

Operano solo tra booleani e producono un booleano. Possono concatenare operazioni di comparazione.

Operatore	Significato
`ET`	AND — vero se entrambi veri
`VEL`	OR — vero se almeno uno è vero
`AUT`	XOR — vero se esattamente uno è vero
`!` (prefisso)	NOT — inverte il booleano

```
#n = 7
IF tra5e10 = @n > 5 ET @n < 10   >> V
IF fuori   = @n < 5 VEL @n > 10  >> F
IF negato  = !@tra5e10            >> F
```

11 Priorità degli operatori

Priorità	Operatori
1 (massima)	`^`
2	`*` `/`
3	`//`
4	`%`
5	`+` `-`
6	`==` `!=`
7	`<=` `>=`
8	`!`
9 (minima)	`ET` `VEL` `AUT`

Le parentesi `()` funzionano come in matematica per forzare un ordine diverso.

12 Operatori incrementali

Scorciatoie per modificare direttamente una variabile numerica:

Operatore	Equivalente a
`var += n`	`var = @var + n`
`var -= n`	`var = @var - n`
`var *= n`	`var = @var * n`
`var /= n`	`var = @var / n`
`var //= n`	`var = @var // n`
`var %= n`	`var = @var % n`
`var ^= n`	`var = @var ^ n`

```
#punteggio = 10
#punteggio += 5   >> ora vale 15
#punteggio *= 2   >> ora vale 30
```

13 Funzioni

Le funzioni sono blocchi di codice che vengono descritti senza essere eseguiti. Si eseguono solo quando vengono chiamati.

13.1 FUN

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

- `parametri` — variabili esterne da passare alla funzione, separati da virgola. Possono essere zero.
- `return` — facoltativo. Se incluso deve stare alla fine del corpo. Il valore di ritorno può essere salvato o stampato. Una sola istruzione `return` per funzione.

```
FUN ciao()
  TALK ciao
  FEND
ciao()
<* Output: ciao *>

nome = Mario
FUN saluta(n)
  TALK ciao @n!
  #num = 5
  return @num
  FEND
val = saluta(nome)
TALK @val
<* Output:
```

```
ciao Mario!
5 *>
```

13.2 THEN

Forma semplificata di funzione: non accetta parametri e non può avere `return`. Si chiude con `THEND`.

```
THEN nome_then
  ...
  THEND
```

Equivale a `FUN nome()` senza `return`. I blocchi THEN possono essere chiamati **solo** all'interno di un GO block.

14 GO Block

GO esegue una funzione o un blocco THEN in base a una condizione booleana.

```
GO @var_condizione @nome_funzione #ripetizioni
```

- `@var_condizione` — variabile IF (o qualsiasi booleano). Il blocco viene eseguito solo se è `V`.
- `#ripetizioni` — quante volte eseguire il blocco. Se `1` si può omettere. Il valore speciale `#c` ripete finché la condizione rimane `V`.

```
IF se = V
THEN allora
  TALK Evviva!
  THEND
GO @se @allora       >> esegue una volta

GO @se @allora #3    >> esegue 3 volte

#i = 0
IF loop = @i < 3
THEN conta
  TALK @i
  #i += 1
  THEND
GO @loop @conta #c   >> ripete finché i < 3
```

14.1 Catene: `&` e `&&`

Dopo un GO si possono aggiungere rami alternativi:

- `&` — equivale a `elif` in Python.
- `&&` — equivale a `else` in Python, si può porre solo alla fine.

```
GO @cond1 @blocco1 & @cond2 @blocco2 && @blocco3
```

```
<* se cond1 → blocco1
   altrimenti se cond2 → blocco2
   altrimenti → blocco3 *>
```

14.2 Corpo inline con `@{}`

Invece di definire un THEN separato, si può scrivere il corpo direttamente nel GO. La condizione può essere anch'essa inline:

```
GO @{@n > 0} @{
  TALK positivo
} && @{
  TALK non positivo
}
```

La variabile speciale `THISGO` contiene la condizione del GO corrente e può essere modificata per interrompere il loop:

```
GO @{V} @{
  TALK @ASKED
  IF stop = @ASKED == fine
  GO @stop @{ @THISGO = F }
}
```

15 Liste

Una lista immagazzina più valori in una sola variabile. I valori sono separati da `;` e racchiusi tra `[]`.

```
lista = [ciao ; salve]
```

Ogni elemento ha un **indice** che parte da 0. Gli indici negativi partono da destra: `-1` è l'ultimo elemento.

Si possono aggiungere **tag** agli elementi con `|`:

```
lista = [saluto | ciao ; salve]
lista = [italiano | saluto | ciao ; salve]  >> ciao ha due tag
```

Per accedere a un elemento:

```
lista[#0]          >> per indice
lista[saluto]      >> per tag
lista[tag1 ; tag2] >> per più tag
```

I valori di una lista sono sempre stringhe.

15.1 Aggiungere elementi

```
ADD nome_lista AT indice = valore   >> inserisce spostando gli altri in avanti
ADD nome_lista AT = valore          >> appende in fondo (indice omesso)
ADD nome_lista BY indice = valore   >> sostituisce l'elemento a quell'indice
```

15.2 Rimuovere elementi

```
CANC nome_lista AT indice    >> per indice
CANC nome_lista BY tag       >> per tag
CANC nome_lista IS valore    >> per valore
CANC nome_lista IN [tag1 ; tag2]  >> per lista di tag
```

> **Attenzione:** se due elementi hanno lo stesso tag o valore, `CANC` li rimuove entrambi.

15.3 Stampa lista

```
lista = [saluto | ciao]
TALK @lista    >> [ciao]
SCREAM @lista  >> [saluto | ciao]
```

Le liste possono contenere altre liste come elementi:

```
dati = [nomi | [Giulio ; Carlo] ; cognomi | [Bianchi ; Ferrari]]
```

16 Scope delle variabili

Per default le variabili dichiarate dentro un blocco sono **locali** — non accessibili dall'esterno.

Prefisso	Effetto
`RET`	Rende la variabile globale
`DEFRET`	Messo all'inizio del blocco, rende **tutte** le variabili del blocco globali per default
`TEMP`	Con `DEFRET` attivo, rende la singola variabile locale

```
FUN calcola()
  RET #risultato = 42   >> accessibile ovunque
  locale = ciao         >> solo dentro questa funzione
  FEND

FUN esempio()
  DEFRET
  #due = 2              >> globale (DEFRET)
  TEMP #quattro = 4     >> locale
  FEND
```

17 Operatore IN

Verifica se un elemento è presente in una lista. Produce un booleano.

```
lista = [ciao ; salve]
IF esiste = ciao IN lista        >> V
IF manca  = arrivederci IN lista >> F
```

18 Ciclo FOR

Serve a iterare gli elementi di una lista.

`FOR @nome_lista #[inizio ; fine ; salto] = @funzione()`

- `inizio` — indice da cui partire. Default: `0`.
- `fine` — indice a cui arrivare. Default: `-1` (fine lista).
- `salto` — passo tra gli indici (positivo o negativo). Default: `1`.

Tutti e tre i valori in `#[]` sono opzionali e il separatore `;` va comunque incluso se si omette solo quello di mezzo.

All'interno della funzione sono disponibili variabili speciali:

Variabile	Contenuto
`ONITEM`	Il valore dell'elemento corrente
`ONINDEX`	L'indice dell'elemento corrente
`ONTAG`	Il/i tag dell'elemento corrente

```
numeri = [10 ; 20 ; 30]
FOR @numeri = @{TALK @ONINDEX: @ONITEM}
<* Output:
0: 10
1: 20
2: 30 *>

>> Iterazione al contrario
FOR @numeri #[;;-1] = @{TALK @ONITEM}
<* Output: 30 / 20 / 10 *>
```

18.1 NUMBERS

Esiste la lista speciale `NUMBERS` che contiene tutti i numeri reali, dove ogni numero ha se stesso come indice. Utile per iterare range numerici:

```
FOR @NUMBERS #[1 ; 5] = @{TALK @ONITEM}
<* Output: 1 / 2 / 3 / 4 / 5 *>
```

19 Etichette

Qualsiasi istruzione può ricevere un'etichetta con la sintassi `parola_chiave:etichetta`. Per rieseguire l'istruzione associata a un'etichetta si usa `*etichetta`.

```
TALK:saluto Ciao!
*saluto   >> riesegue il TALK:saluto
```

Se a una stessa etichetta vengono assegnate più istruzioni, `*etichetta` esegue l'ultima.

19.1 Etichette speciali di TALK

`TALK:V` — attiva un input persistente a fine output. L'input viene salvato nella variabile speciale `ASKED` (vedi sezione 7).

`TALK:F` — disattiva l'input persistente.

20 Gestione errori

20.1 Codici di errore

Codice	Causa
`VAR_NOT_FOUND`	Variabile inesistente
`TYPE_ERROR`	Tipo di dato non valido
`INVALID_CONVERSION`	Conversione di tipo non possibile
`CONST_MODIFY`	Tentativo di modifica di una costante (STAY)
`DIV_BY_ZERO`	Divisione per zero
`MATH_ERROR`	Operazione matematica non valida
`BOOL_EXPECTED`	Atteso valore booleano
`NUMBER_EXPECTED`	Atteso valore numerico
`STRING_EXPECTED`	Attesa stringa
`LIST_NOT_FOUND`	Lista inesistente
`LIST_OUT_OF_RANGE`	Indice fuori dai limiti della lista
`LIST_EMPTY`	Lista vuota
`TAG_NOT_FOUND`	Tag inesistente nella lista
`DUPLICATE_TAG`	Tag duplicato non consentito
`FUNC_NOT_FOUND`	Funzione inesistente
`INVALID_ARGUMENTS`	Argomenti non validi
`RETURN_NOT_ALLOWED`	Uso di `return` non consentito
`THEN_ARGUMENTS`	Uso di argomenti in un blocco THEN
`IF_NOT_FOUND`	IF block inesistente
`GO_INVALID`	GO block non valido
`GO_LOOP_ERROR`	Loop infinito o non valido
`INPUT_ERROR`	Errore nell'input
`OUTPUT_ERROR`	Errore di output
`SYNTAX_ERROR`	Errore di sintassi
`ESCAPE_ERROR`	Escape non valido
`MOD_NOT_FOUND`	Modulo non trovato
`UNKNOWN_ERROR`	Errore sconosciuto

20.2 TRY / SHOW / YET

Permettono di gestire errori di esecuzione senza interrompere bruscamente il programma.

```
TRY nome_try
  ...codice che potrebbe dare errore...
  TREND

SHOW nome_show @nome_try
  ...eseguito solo se c'è stato un errore...
```

```
  SEND

YET @nome_try
  ...eseguito sempre (con o senza errore)...
  YEND
```

`SHOW` e `YET` sono facoltativi.

All'interno di `SHOW` sono disponibili due variabili speciali — esistono **solo** lì:

Variabile	Tipo	Contenuto
`ERRCODE`	numerale	`1` per tutti gli errori gestiti da LINE
`ERRMSG`	stringa	Messaggio descrittivo dell'errore

```
TRY prova
  OUT @inesistente
  TREND

SHOW gestisci @prova
  TALK @ERRCODE
  TALK @ERRMSG
  SEND

YET @prova
  TALK Fine operazione
  YEND
```

20.3 Regole

- `TRY` non può essere annidato.
- Ogni `TRY` può avere al massimo un `SHOW` e un `YET`.
- Se un errore avviene in un `TRY` senza `SHOW`, LINE stampa `ERRMSG` e termina l'esecuzione.

21 Moduli

I moduli sono collezioni di funzioni pronte all'uso. Si importano con `TAKE`:

```
TAKE math
#r = math_sqrt(9)
TALK @r   >> 3
```

Le funzioni di un modulo si chiamano con il formato `nomemodulo_nomefunzione()`.

Per vedere l'elenco di tutti i moduli disponibili, esegui `.MODS` in un file vuoto. Per vedere le funzioni di un modulo specifico, usa `.MATH`, `.STR`, ecc.

Modulo	Contenuto
`math`	Funzioni matematiche (sqrt, abs, floor, ceil, pow, sin, cos...)
`str`	Manipolazione stringhe (len, sub, upper, lower, replace...)
`list`	Operazioni su liste (len, has, sort, rev, shuffle...)
`rand`	Casualità (int, float, pick, shuffle, dice...)
`io`	Input/output avanzato (time, date...)
`type`	Verifica e conversione di tipi
`conv`	Conversioni (hex, bin, oct, tonum...)
`num`	Funzioni numeriche avanzate (fmt, fib, fact, gcd, prime...)

22 File LINE

Un programma LINE viene salvato con estensione `.line` ed eseguito dall'interprete.

23 LINE-DOM

LINE-DOM è un layer JavaScript che permette di usare LINE come linguaggio per controllare il DOM di una pagina HTML, senza scrivere JavaScript. Si include con il file `line-dom.js`.

```
<!-- Interprete LINE -->
<script src="interpreter.js"></script>
<!-- Bridge DOM -->
<script src="line-dom.js"></script>
```

> **Regola fondamentale:** nei tag `<script type="text/line">` va *solo* codice LINE. Zero JavaScript inline.

23.1 Attributi del tag `<script type="text/line">`

Attributo	Effetto
`trigger="id"`	Esegue il codice al click sull'elemento con quell'id
`autorun`	Esegue il codice al caricamento della pagina
`replace`	Sovrascrive il contenuto dell'elemento target invece di aggiungerlo
`counter="id"`	Inietta la variabile speciale `__N__` (numero di esecuzioni) e la scrive nell'elemento indicato
`body-class="id"`	Osserva l'elemento indicato e applica il suo testo come `className` del `<body>` (utile per temi CSS)
`onclick="btnId:label,..."`	Collega il click di `#btnId` al blocco `label`

```
<script type="text/line" trigger="btn-invia" replace>
  risposta = Hai cliccato!
  <risultato>TALK @risposta</risultato>
```

```
</script>

<button id="btn-invia">Invia</button>
<div id="risultato"></div>
```

23.2 Scrivere su elementi HTML

Si racchiude il codice LINE tra tag con l'id dell'elemento di destinazione:

Sintassi	Effetto
`<id>TALK testo</id>`	Sostituisce il contenuto dell'elemento
`<id=after>TALK testo</id>`	Appende in fondo al contenuto esistente
`<id=before>TALK testo</id>`	Inserisce all'inizio del contenuto esistente
`<.classe>TALK testo</.classe>`	Scrive su tutti gli elementi con quella classe CSS
`<.classe=after>TALK testo</.classe>`	Appende a tutti gli elementi con quella classe

Il carattere `\n` dentro i tag DOM diventa automaticamente un `<br>` nell'HTML (non vale per gli input).

```
<p id="saluto"></p>

<script type="text/line" trigger="btn" replace>
  <saluto>TALK Ciao, mondo!</saluto>
</script>
```

23.3 Leggere input HTML

LINE-DOM inietta automaticamente il valore di ogni `<input>`, `<textarea>` e `<select>` che ha un `id` come variabile LINE prima di ogni esecuzione. Non serve nessuna dichiarazione.

```
<input id="nome" type="text">
<button id="btn">Saluta</button>
<div id="output"></div>

<script type="text/line" trigger="btn" replace>
  <output>TALK Ciao, @nome!</output>
</script>
```

Gli input hidden funzionano come memoria persistente tra un'esecuzione e l'altra.

23.4 Gestire click da codice LINE

Si può registrare il click su un elemento direttamente nella sintassi LINE:

Sintassi	Significato
`THEN:btn-id nomeblocco`	Il blocco THEN viene eseguito al click su quell'id
`GO:btn-id @{cond} @{...}`	GO inline eseguito al click

```
<script type="text/line" autorun>
  THEN:btn-maiuscolo
    <output>TALK CIAO</output>
    THEND
</script>
```

> **Nota:** i tag `<id>` dentro un corpo `@{...}` inline non funzionano. Per blocchi che scrivono su elementi, usa sempre un THEN con nome.

23.5 LocalStorage con `<ls>`

LINE-DOM supporta la persistenza dei dati tra ricaricamenti tramite un tag speciale:

```
<ls>nomevar = valore</ls>
```

Salva `valore` nel localStorage con chiave `line_nomevar`. Ad ogni esecuzione successiva, il valore viene iniettato automaticamente come variabile LINE.

```
<input id="nome">
<button id="btn-salva">Salva</button>
<div id="benvenuto"></div>

<script type="text/line" autorun>
  <benvenuto>TALK Ciao, @nome!</benvenuto>
</script>

<script type="text/line" trigger="btn-salva">
  <ls>nome = @nome</ls>
</script>
```

23.6 Temi dinamici con `body-class`

L'attributo `body-class` crea un `MutationObserver`: osserva l'elemento indicato e applica automaticamente il suo testo come classe del `<body>`. Combinato con variabili CSS, permette di cambiare tema senza JavaScript.

```
<span id="tema" hidden></span>

<script type="text/line" trigger="btn" replace body-class="tema">
  <tema>TALK dark</tema>
</script>
```

```
body.dark  { --bg: #111; --fg: #eee; }
body.light { --bg: #fff; --fg: #111; }
```

23.7 Comportamento del re-run

Ogni volta che il codice viene eseguito (ad esempio al click di un trigger), l'intero script viene ri-eseguito dall'inizio. Questo significa che le variabili si resettano a ogni esecuzione. Per la persistenza dello stato usare `<ls>` (localStorage) o input hidden.

24 LINE-RENDER

LINE-RENDER è un'estensione dell'IDE che permette di visualizzare formule matematiche e strutture chimiche nella console. Richiede `line-render.js` e la libreria KaTeX.

La notazione è una versione abbreviata che viene compilata in LaTeX prima del rendering.

24.1 Blocchi RENDER

Sintassi per il rendering a blocco (su una riga separata, centrato):

```
RENDER{
  formula
}
```

Tutto ciò che è tra `{` e `}` viene interpretato come formula e renderizzato nella console.

```
RENDER{
  s(k=0,n)(k2) = n(n+1)/2
}

RENDER{
  de(f)/de(x) = lim(h -> 0)((f(x+h) - f(x))/h)
}
```

24.2 Render inline `\R...\R`

Per includere una formula direttamente all'interno di una riga di testo si usa la coppia di escape `\R`:

```
TALK Questo è un integrale \RS(0,1)(f(x))\R, ed è molto interessante.
TALK La formula di Eulero è \Reu^(pi*i) + 1 = 0\R.
```

La formula tra i due `\R` viene renderizzata con KaTeX in modalità inline, allineata verticalmente al testo circostante.

24.3 Modalità CHEM — Strutture chimiche

Per disegnare molecole la prima riga del blocco deve essere `CHEM`. Ogni riga successiva descrive una molecola.

```
RENDER{
  CHEM
  C(HHH=O)
}
```

Sintassi delle molecole:

- Atomi: simboli chimici come `C`, `H`, `O`, `N`, `Cl`, ecc.
- Legami impliciti: `-` (singolo, default)
- Legame doppio: `=`
- Legame triplo: `#`

- Ramificazioni: `(...)` per i sostituenti

```
RENDER{
  CHEM
  C(HHH=O)          >> metanale (formaldeide)
}

RENDER{
  CHEM
  C(H)(H)(=O)       >> carbonile con due H
}
```

24.4 Notazione matematica

La notazione è compatta. Ogni espressione viene tradotta in LaTeX.

24.4.1 Operazioni di base

Notazione	Rendering
`a + b`	$a + b$
`a - b`	$a - b$
`a * b`	$a \cdot b$
`a/b`	ab
`(a+b)/(c+d)`	$a + bc + d$
`n!`	$n!$

24.4.2 Potenza e pedice

Notazione	Rendering
`x5`	x^5
`x^(n+1)`	x^{n+1}
`x#(i)`	x_i
`a#(i)^(2)`	a_i^2

24.4.3 Radici

Notazione	Rendering
`R(x2+1)`	$\sqrt{x^2 + 1}$
`R3(a3)`	$\sqrt[3]{a^3}$

24.4.4 Derivate

Notazione	Tipo
`f'`, `y''`, `x'''`	Notazione di Newton
`de(f)/de(x)`	Derivata ordinaria $df dx$
`dp(f)/dp(x)`	Derivata parziale $\partial f \partial x$

24.4.5 Sommatorie, produttorie, integrali, limiti

Notazione	Rendering
`s(k=0,n)(k2)`	$\sum_{k=0}^{n} k^2$
`P(k=1,n)(k)`	$\prod_{k=1}^{n} k$
`S(0,1)(x2)`	$\int_0^1 x^2\,dx$
`L(0)((sin(x))/x)`	$\lim_{x\to 0} \sin x x$

24.4.6 Funzioni standard

Notazione	Funzione
`sin(x)`, `cos(x)`, `tan(x)`	Trigonometriche
`arcsin(x)`, `arccos(x)`, `arctan(x)`	Arco-trigonometriche
`sinh(x)`, `cosh(x)`, `tanh(x)`	Iperboliche
`ln(x)`, `log(x)`, `exp(x)`	Logaritmi ed esponenziale

24.4.7 Vettori e matrici

Notazione	Rendering
`v(x;y;z)`	Vettore colonna
`M(2;2)(a;b;c;d)`	Matrice 2×2

24.4.8 Combinatoria

Notazione	Rendering
`C(n;k)`	nk
`Perm(n;k)`	A_n^k
`CombR(n;k)`	$n+k-1k$

24.4.9 Valori assoluti e norme

Notazione	Rendering
`|x - a|`	$\|x-a\|$
`||v||`	$\|\|v\|\|$

24.4.10 Decoratori

Notazione	Effetto
`vc(v)`	Vettore $\vec{v}$
`ha(x)`	Cappello $\hat{x}$
`ba(x)`	Barra $\bar{x}$
`ti(x)`	Tilde $\tilde{x}$
`do(x)`	Punto $\dot{x}$
`do2(x)`	Doppio punto $\ddot{x}$
`fl(x)`	Floor $\lfloor x \rfloor$
`ce(x)`	Ceiling $\lceil x \rceil$

24.4.11 Costanti e lettere greche

Notazione	Simbolo
`pi`	π
`eu`	e
`I`	∞
`na`	∇
`alp`, `bet`, `gam`, `del`, `eps`	$\alpha\ \beta\ \gamma\ \delta\ \varepsilon$
`the`, `lam`, `mu`, `sig`, `phi`, `ome`	$\theta\ \lambda\ \mu\ \sigma\ \varphi\ \omega$
`Gam`, `Sig`, `Phi`, `Psi`, `Om`	$\Gamma\ \Sigma\ \Phi\ \Psi\ \Omega$

24.4.12 Relazioni e logica

Notazione	Simbolo
`->`	$\rightarrow$
`=>`	$\Rightarrow$
`<=>`	$\Leftrightarrow$
`~`	$\approx$
`!=`	$\neq$
`<=`, `>=`	$\leq$, $\geq$

24.5 Salvataggio PDF della console

Il bottone **PDF** nella barra della console genera un PDF visivo della console, conservando il rendering delle formule e dei grafici chimici. Il PDF viene ridimensionato al contenuto effettivo.

Il bottone **TXT** salva invece il testo puro della console (le formule appaiono come testo sorgente).

24.6 Modalità HTML

La prima riga del blocco può essere `HTML`. Tutto il contenuto successivo viene interpretato come HTML grezzo e inserito direttamente nella console tramite `innerHTML`.

```
RENDER{
  HTML
  ...codice HTML...
}
```

Questo permette di visualizzare nella console qualsiasi struttura HTML: tabelle, liste, immagini, testo stilizzato, componenti arbitrari.

```
RENDER{
  HTML
  <table border="1" style="border-collapse:collapse;color:#cdd9e5">
    <tr><th>Nome</th><th>Valore</th></tr>
    <tr><td>pi</td><td>3.14159</td></tr>
    <tr><td>e</td><td>2.71828</td></tr>
  </table>
}
```

```
RENDER{
  HTML
  <ul style="color:#80cbc4">
    <li>Primo elemento</li>
    <li>Secondo elemento</li>
  </ul>
}
```

> **Nota:** il contenuto HTML viene inserito senza sanitizzazione. Usare tag di script all'interno non è consigliato — l'HTML viene renderizzato nel contesto della console dell'IDE.