ATOMIC

Parti a razzo a programmare!!

v 1.0.02 - 11/05/2017

MANIFESTO

ATOMIC è un linguaggio di programmazione a **scopo didattico**.
È stato progettato per rispettare queste caratteristiche:

- Realmente facile da imparare, soprattutto per chi non ha mai programmato
- Esplicito e intuitivo, facilmente leggibile e prossimo all'italiano parlato
- Flessibile, tollerante ma preciso

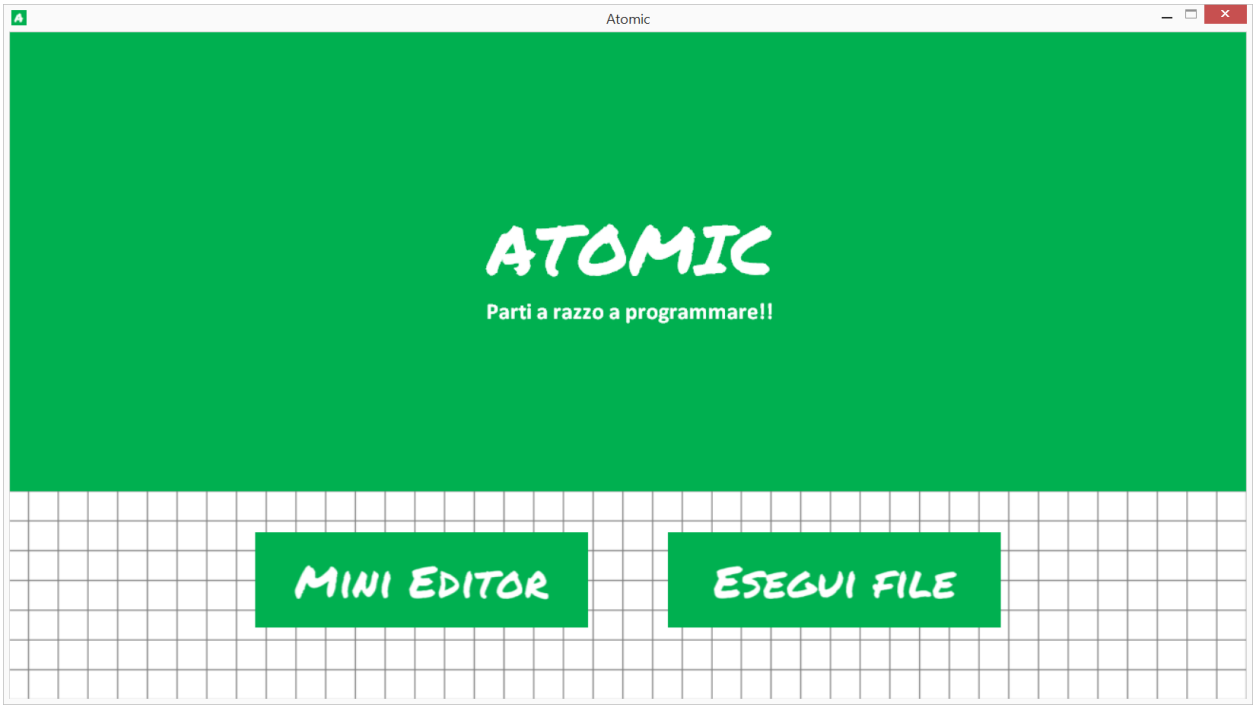
ATOMIC non è uno strumento per creare progetti a lungo termine, è stato pensato come una "propulsione esplosiva" che permette di "partire a razzo" con la programmazione a discapito della longevità di utilizzo. Se dei linguaggi di programmazione "seri" come C++, Java o Python sono paragonabili a dei pratici e multifunzionali coltellini svizzeri, ATOMIC è paragonabile ad una brutale ascia da battaglia: pesante, ingombrante, un po' rozzo ma incredibilmente efficace nello svolgere in modo semplice i pochi compiti per i quali è stato pensato!

Questo documento è concepito come guida rivolta ai docenti. Per la comprensione non sono necessarie particolari conoscenze informatiche per quanto riguarda la programmazione ma solo conoscenze base d'informatica e matematica.

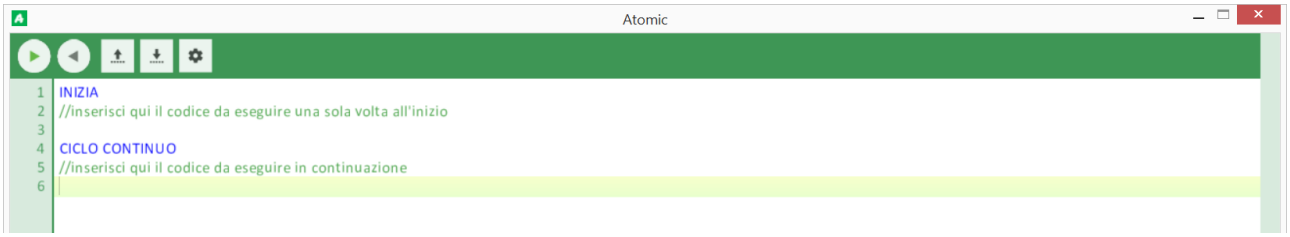
Buona lettura!

AMBIENTE DI SVILUPPO INTEGRATO E NOTEPAD ++

Attualmente ATOMIC è disponibile solo per Windows ed è scaricabile da questo indirizzo: http://www.bergame.eu/Atomic.rar .
Il pacchetto contiene solo quattro cose: questa guida, l'interprete, il file .xml per Notepad++ e una cartella con alcuni esempi.
Avviando l'interprete di ATOMIC (Atomic.exe) si hanno a disposizione solo due opzioni: **Mini editor** ed **Esegui file**.

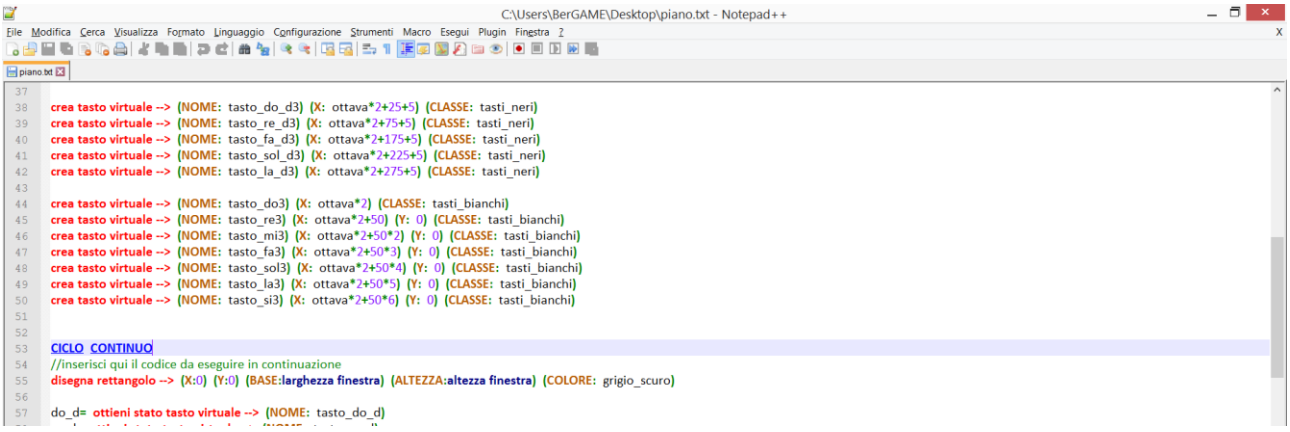


Cliccando su **Esegui file** si scegli direttamente il file di testo da eseguire (.txt).
Cliccando su **Mini editor** si apre un piccolo IDE (ambiente di sviluppo integrato):



Questa applicazione interna è molto pratica e minimale ma è consigliata solo per testare brevi pezzi di codice. Possiede solo le funzionalità base di un semplice editor di testo (carica, salva, annulla con CTRL+Z). **Il suo utilizzo è altamente sconsigliato quando si lavora con un codice lungo oltre le 30-40 righe.**

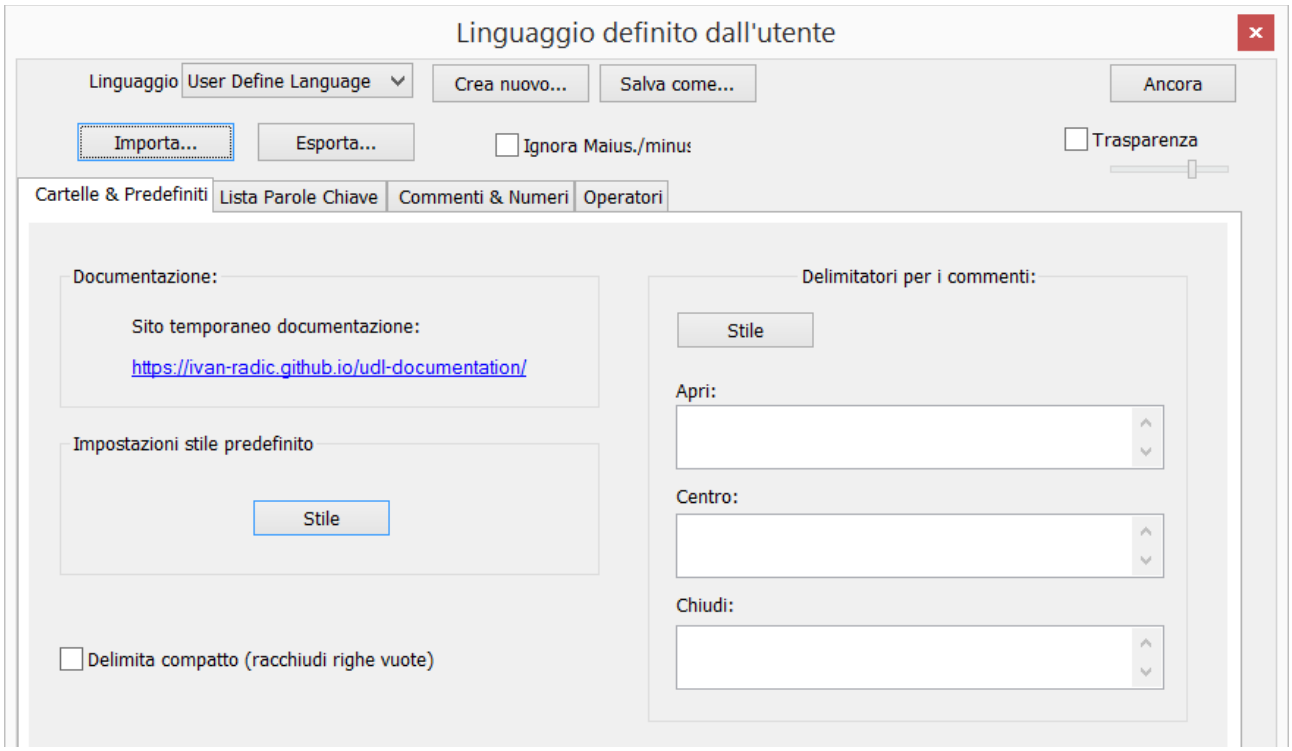
Per lavorare con del codice corposo è consigliabile l'utilizzo di Notepad++



Notepad++ è un editor di testo gratuito e open source per sistemi MS Windows, che supporta diversi linguaggi di programmazione.

È scaricabile gratuitamente a questo indirizzo: https://notepad-plus-plus.org/download/v7.3.3.html .

Una volta scaricato è possibile importare il file che definisce il linguaggio ATOMIC (Atomic.xml).
La procedura è molto semplice: *Linguaggio -> Definisci il tuo linguaggio… -> Importa… -> Atomic.xml*



Dopo averlo importato chiudere Notepadd++. Riaprendolo la voce “Atomic” comparirà in fondo al menu “Linguaggio”. Selezionate “Atomic” per formattare il testo secondo la sintassi di Atomic.

Se non si vuole scaricare Notepad++ è possibile scrivere codice ATOMIC semplicemente con il **Blocco note**, l’unico svantaggio è l’assenza di formattazione automatica (evidenziazione) del codice.

I file di Atomic sono semplici file di testo con estensione *.txt*.

EVENTI

Gli eventi indicano quando una determinata parte del codice dovrà essere eseguita.

Esistono solo due eventi in ATOMIC: **INIZIA** e **CICLO CONTINUO**.

L'evento INIZIA viene eseguito una sola volta quando inizia il programma.

L'evento CICLO CONTINUO viene eseguito subito dopo INIZIA e continua ad essere eseguito ogni 1/30 secondi (ogni trentesimo di secondo) finché il programma è in esecuzione.

La sintassi da utilizzare è la seguente:

```
INIZIA
… codice …

CICLO CONTINUO
… codice ...
```

ESPRESSIONI

Un'espressione è un costrutto che utilizza numeri, variabili, costanti combinandoli con gli operatori per restituire un risultato.

Gli operatori utilizzabili sono i seguenti:

OPERATORE	DESCRIZIONE
+	Somma
-	Sottrazione
*	Moltiplicazione
/	Divisione
^	Potenza

Esempi:

2+2

somma due più due, il risultato è 4

3*2+2

moltiplica tre per due e poi somma due, il risultato è 8

3*2+2*5

moltiplica tre per due e poi somma due per dieci, il risultato è 16

5^2

eleva cinque alla seconda, il risultato è 25

È anche possibile utilizzare le parentesi tonde "( )" per modificare l'ordine di esecuzione dei calcoli.

Ad esempio:

3*(2+2)*5

da come risultato 60 invece che 16

L'ordine in cui vengono eseguiti i calcoli è uguale a quello convenzionale della matematica:

1° - vengono calcolate tutte le sotto espressioni nelle parentesi, a partire da quelle più interne

2° - vengono calcolati gli elevamenti a potenza

3° - vengono calcolate le moltiplicazioni e le divisioni

4° - vengono calcolate le addizioni e le sottrazioni

Esempio:

((2+3*2)/2)^(2+1)

$$\left(\frac{2+3\cdot 2}{2}\right)^{2+1}$$

da come risultato 64:

= ((2+6)/2)^(2+1) = (8/2)^(2+1) = 4^(2+1) = 4^3 = 64

ovvero:

$$= \left(\frac{2+6}{2}\right)^{2+1} =$$

$$= \left(\frac{8}{2}\right)^{2+1} =$$

$$= \left((2)^2\right)^{2+1} =$$

$$= \left((2)^2\right)^{3} =$$

$$= (2)^6 =$$

$$= 64$$

Come in matematica è anche possibile utilizzare costanti e variabili (vedi più avanti) all'interno di un'espressione

Esempi:

100*pi greco
2+gatto
(cane*5/2)+topo^2

Ad esempio in matematica *(cane*5/2)+topo^2* si potrebbe scrivere in questo modo:

$$\left(c \cdot \frac{5}{2}\right) + t^2$$

Dove *c* sta per cane e *t* per topo.

Nel caso serva, per migliorare la leggibilità è possibile inserire <u>uno</u> spazio tra gli elementi di una espressione.

Ad esempio le seguenti forme di scrittura sono valide:

((2 + 3 * 2) / 2) ^ (2 + 1)
((2+3*2) / 2) ^ (2+1)
((2+3*2)/2)^(2+1)

Un numero che contiene una parte decimale può essere scritto utilizzando il punto (.) o la virgola (,).

Esempio:

5,23
5.23

Sono entrambe scritture valide.

Un'espressione può essere utilizzata all'interno di altri costrutti, come valore di una variabile o come argomento di una funzione.

VARIABILI

Le variabili sono locazioni di memoria che contengono delle informazioni modificabili. Le variabili hanno un nome in modo che è possibile averne un riferimento. Una variabile in ATOMIC può contenere sia un numero reale che una stringa (una riga di testo). Esistono anche variabili integrate come ad esempio ***x del mouse*** e ***y del mouse*** che indicano la posizione del cursore del mouse. Prima di utilizzare una variabile bisogna dichiararla. Normalmente è buona prassi dichiararle nell'evento **INIZA**, tuttavia possono anche essere dichiarate nell'evento **CICLO CONTINUO** (in alcuni casi è più vantaggioso).
Per modificare e dichiarare una variabile si utilizza la stessa sintassi, ovvero:

nome = valore

Ci sono vari modi per dichiarare o modificare il valore una variabile:

Tramite valore reale, es:

```
gatto = 1
gatto = 2.54
```

Tramite stringa di testo, es:

```
gatto = "Ciao gatto!"
```

Tramite costante, es:

```
gatto = vero
gatto = pi greco
mio_colore = verde
```

Tramite altra variabile, es:

```
cane = 1
gatto = cane
```

Tramite espressione, es:

```
gatto = 2+2
gatto = 5*2+(1+5/2)-6^2
gatto = cane*pi greco
```

Tramite funzione che restituisce un valore (tutte quelle che iniziano con "ottieni"), es:

```
gatto = ottieni uno a caso di questi valori -->  (VALORE 1 : 50 ) (VALORE 2 : 100 ) (VALORE 3 : 70 )
gatto = ottieni il logaritmo naturale di -->  (VALORE : 324 )
mio_colore = scegli tra questi valori -->  (VALORE 1 : verde chiaro ) (VALORE 2 : verde oliva ) (VALORE 3 : verde acqua )
```

Una variabile può contenere due tipi di valore: **numero reale** o **stringa di testo**.

Nel corso della sua esistenza una variabile può cambiare il tipo di valore che contiene (tipizzazione dinamica).
Ad esempio:

```
INIZIA
gatto = 50
tempo = 0

CICLO CONTINUO
aumenta tempo_trascorso di 1
se tempo_trascorso < 100 allora aumenta gatto di 0.5     disegna cerchio --> (RAGGIO: gatto) .
se tempo_trascorso >= 100 allora gatto = "Sono un gatto!"     disegna testo --> (TESTO: gatto) .
```

è una scrittura valida.

E' possibile gestire gli spazi tra il simbolo "=" come si preferisce, le seguenti forme di scrittura sono valide:

```
gatto=1
gatto = 1
```

```
gatto= 1
gatto =1
gatto=    1
gatto     =     1
gatto
=
1
```

Solo le variabili integrate come *x del mouse* **possono avere un nome separato da spazi " ".**
Le variabili definite dall'utente se formate da due o più parole possono essere divise dal simbolo "_".

Esempio corretto:

```
mio_colore = verde
//CORRETTO!
```

Esempio sbagliato:

```
mio colore = verde
//SBAGLIATO!!!
```

ELENCO DELLE VARIABILI INTEGRATE

Le variabili integrate sono delle variabili già presenti in ATOMIC. Non è necessario dichiararle.
Alcune possono essere modificate manualmente, altre vengono gestite automaticamente e possono essere solamente lette.

NOME VARIABILE	DESCRIZIONE	MODIFICABILE MANUALMENTE?
x del mouse	La posizione in pixel sull'asse x del cursore (freccetta) del mouse	no
y del mouse	La posizione in pixel sull'asse y del cursore (freccetta) del mouse	no
larghezza finestra	La larghezza in pixel della finestra	si
altezza finestra	L'altezza in pixel della finestra	si
colore sfondo	Il colore dello sfondo della finestra	si
nome finestra	Il nome della finestra visualizzato in alto (stringa)	si
tasto "nome"* è stato premuto	Il risultato della verifica se il tasto "nome"* è stato premuto (vero o falso)	no
tasto "nome"* è stato rilasciato	Il risultato della verifica se il tasto "nome"* è stato rilasciato (vero o falso)	no
tasto "nome"* è premuto	Il risultato della verifica se il tasto "nome"* è attualmente premuto (vero o falso)	no
rotella del mouse in su	Il risultato della verifica se la rotella del mouse è stata ruotata in su (vero o falso)	no
rotella del mouse in giu	Il risultato della verifica se la rotella del mouse è stata ruotata in giu (vero o falso)	no

* "nome" può essere una delle seguenti parole: invio, barra spaziatrice, indietro, ctrl, alt, freccia su, freccia giu, freccia sinistra, freccia destra, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, Q, W, E, R, T, Y, U, I, O, P, A, S, D, F, G, H, J, K, L, Z, X, C, V, B, N, M, sinistro del mouse, destro del mouse, centrale del mouse.

FUNZIONI

Una funzione è un costrutto che esegue automaticamente un'operazione complessa utilizzando determinati argomenti (parametri) forniti dal programmatore.

Le funzioni in ATOMIC hanno questa forma:

nome della funzione --> (ARGOMENTO 1: …) (ARGOMENTO 2: …) …

Normalmente nei linguaggi di programmazione gli argomenti di una funzione vanno forniti in un certo ordine e tutti gli argomenti necessari vanno forniti.

Esempio in pseudocodice:

```
nome_funzione(argomento1,argomento2,argomento3);
```

Un esempio concreto di un altro linguaggio:

```
draw_circle(100,300,200,0);
```

Questa funzione (di un altro linguaggio, il gml, uno tra i più facili da imparare) disegna un cerchio alla posizione x 100, y 300 con 200 pixel di raggio.
A parte l'inglese, per un neofita è difficile capire a cosa si riferiscono quei numeri senza guardare un manuale o perlomeno senza un suggerimento come "*draw_circle(x,y,r,outline);".*

In ATOMIC invece gli argomenti delle funzioni hanno un'*etichetta* e possono essere dati in un ordine casuale.

La funzione vista qui sopra scritta in ATOMIC diventa:

```
disegna cerchio --> (X: 100) (Y: 300) (RAGGIO: 200)
```

che può anche essere scritta senza problemi in questo modo:

```
disegna cerchio --> (Y: 300) (RAGGIO: 200) (X: 100)
```

Inoltre è possibile aggiungere in modo chiaro altri argomenti supportati dalla funzione:

```
disegna cerchio --> (Y: 300) (RAGGIO: 200) (X: 100) (COLORE:  blu)
```

è anche possibile non specificare degli argomenti (anche se fondamentali), ad esempio:

```
disegna cerchio --> (COLORE:  blu)
```

disegnerà un cerchio blu di una dimensione imprecisata in un punto imprecisato.

```
disegna cerchio
```

disegnerà un cerchio nero (colore di base) di una dimensione imprecisata in un punto imprecisato.

Una funzione può anche avere zero argomenti supportati (nessun argomento è richiesto per il suo funzionamento).

Alcune funzioni non restituiscono alcun valore ma svolgono semplicemente un lavoro (ad esempio *disegna cerchio*) altre invece, tutte quelle che iniziano con "ottieni", restituiscono un risultato che può essere memorizzato in una variabile (ad esempio *ottieni la media tra*).

L'argomento di una funzione può essere un numero reale, una stringa (riga di testo), un'espressione, una variabile o una costante.

Funzioni di disegno di forme geometriche

Le funzioni di disegno sono le più facili da apprendere, le uniche nozioni richieste per il loro utilizzo sono le basi della geometria euclidea e del piano cartesiano. Inoltre offrono un primo approccio molto concreto al linguaggio.

disegna cerchio --> (X:) (Y:) (RAGGIO:) (COLORE:) (TRASPARENZA:) (SOLO CONTORNO:)

disegna ellisse --> (X 1:) (Y 1:) (X 2:) (Y 2:) (COLORE:) (TRASPARENZA:) (SOLO CONTORNO:)

disegna rettangolo --> (X:) (Y:) (BASE:) (ALTEZZA:) (COLORE:) (TRASPARENZA:) (SOLO CONTORNO:)

disegna linea --> (X 1:) (Y 1:) (X 2:) (Y 2:) (SPESSORE:) (COLORE:) (TRASPARENZA:)

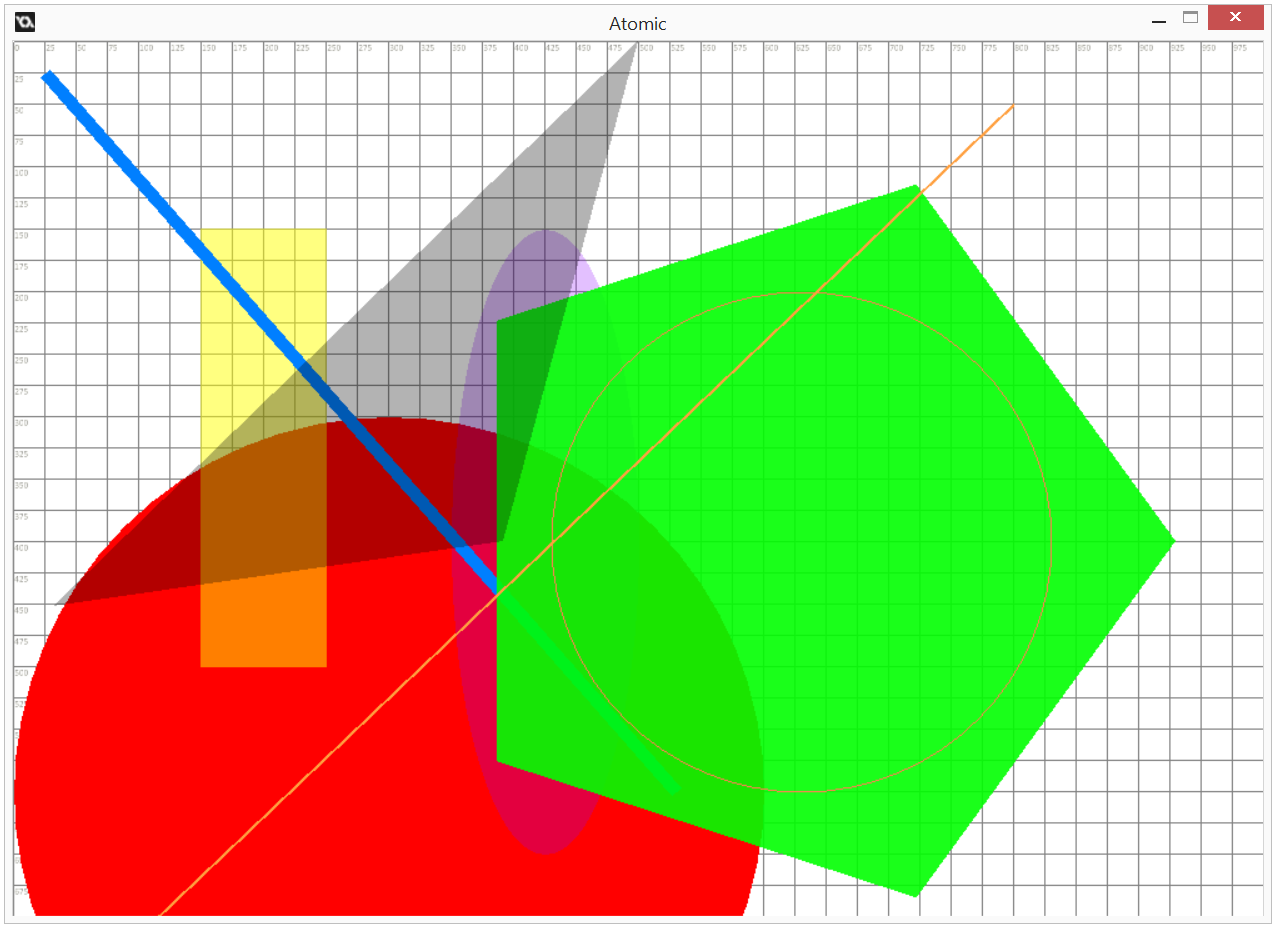
disegna punto --> (X:) (Y:) (COLORE:) (TRASPARENZA:)

disegna poligono regolare --> (X:) (Y:) (NUMERO LATI:) (RAGGIO:) (ROTAZIONE:) (COLORE:) (TRASPARENZA:) (SOLO CONTORNO:)

disegna triangolo --> (X 1:) (Y 1:) (X 2:) (Y 2:) (X 3:) (Y 3:) (COLORE:) (TRASPARENZA:) (SOLO CONTORNO:)

Ecco un esempio in cui vengono utilizzate queste funzioni:

disegna cerchio --> (X: 300) (Y: 600) (RAGGIO: 300) (COLORE: rosso)
disegna rettangolo --> (X 1: 200) (Y 1: 20) (BASE: 100) (ALTEZZA: 350) (RAGGIO: 300) (COLORE: giallo) (TRASPARENZA: 0.5)
disegna ellisse --> (X 1: 350) (Y 2: 253) (X 2: 500) (Y 2: 650) (COLORE: viola) (TRASPARENZA: 0.25)
disegna linea --> (X 1: 25) (Y 1: 25) (X 2: 530) (Y 2: 600) (COLORE: azzurro) (SPESSORE: 10)
disegna poligono regolare --> (NUMERO LATI: 5) (RAGGIO: 300) (X: 630) (Y: 400) (COLORE: verde) (TRASPARENZA: 0.9)
disegna cerchio --> (RAGGIO: 200) (X: 630) (Y: 400) (COLORE: corallo) (TRASPARENZA: 0.5)
disegna linea --> (X 1: 800) (Y 1: 50) (X 2: 10) (Y 2: 800) (COLORE: arancione) (SPESSORE: 2)
disegna triangolo --> (X 1: 500) (X 2: 32) (X 3: 392) (Y 1: 0) (Y 2: 452) (Y 3: 400) (COLORE: nero) (TRASPARENZA: 0.3)



Alcuni aspetti che richiedono un chiarimento:

1) **l'origine della finestra** (il piano cartesiano) come in altri ambiti informatici è **in alto a sinistra** e non in basso a sinistra come da convenzione matematica. Questo ribaltamento dell'asse y è comodo soprattutto quando si lavora con il testo.

2) L'unità per esprimere le dimensioni in Atomic è una sola: il **pixel**. La griglia standard è composta da quadratini di 25 pixel.

3) l'argomento TRASPARENZA deve contenere un valore **compreso tra 0 e 1**.
0=completamente trasparente, 1=completamente visibile, 0.5=mezzo trasparente. Questo range di valori (logica fuzzy) è molto utilizzato sia in Atomic che in generale nell'informatica. È anche possibile utilizzare costanti come *niente*(0), intera(1), mezza(0.5), *un quarto*(0.25), ecc… (vedi sezione COSTANTI).

4) I colori visualizzati sul monitor in realtà sono numeri ma sono esprimibili tramite **costanti** (rosso, verde, giallo, blu, ecc…) vedi la sezione COSTANTI per la lista di tutti i colori disponibili. È anche possibile creare colori personalizzati (vedi più avanti).

5) tutte le funzioni di disegno vanno utilizzate nell'evento **CICLO CONTINUO**. Questo può sembrare contro intuitivo ma la spiegazione è semplice: lo schermo del computer è come una lavagna che ogni trentesimo di secondo viene cancellata. Se ciò non avvenisse sarebbe impossibile creare l'illusione delle animazioni. Immaginate di disegnare molte cose sempre sullo stesso foglio e sempre nella stessa posizione, dopo un po' quello che apparirà sarà solo un ammasso di colore scuro e informe!

Funzioni di disegno del testo

Queste funzioni servono per visualizzare dei testi all'interno del programma che si vuole creare. La funzione principale è ***disegna testo***; questa funzione può essere molto semplice e immediata da utilizzare ma può anche essere arricchita da molti argomenti per personalizzare la visualizzazione del testo. Per introdurla è consigliabile utilizzare solo gli argomenti X, Y, TESTO e COLORE .

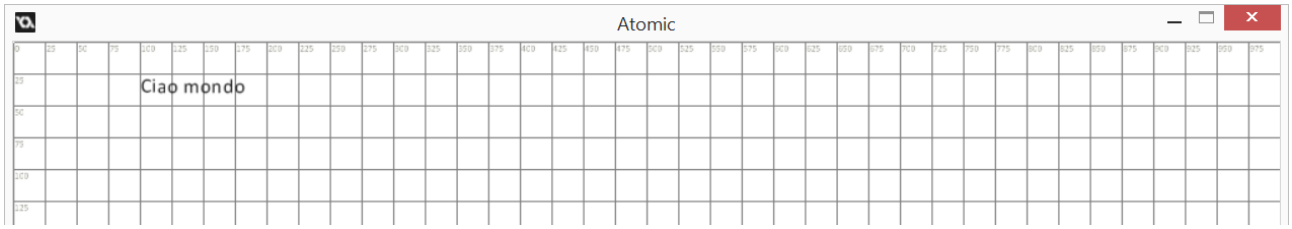
disegna testo --> (X:) (Y:) (TESTO:) (CARATTERE:) (SCALA:) (COLORE:) (TRASPARENZA:) (ROTAZIONE:)
(LARGHEZZA CASELLA:) (INTERLINEA:) (CARATTERE:) (ALLINEAMENTO ORIZZONTALE:)
(ALLINEAMENTO VERTICALE:) (STILE:)

L'argomento TESTO a differenza di quelli visti fino ad ora deve essere una **stringa** ovvero un testo racchiuso tra i simboli " ".Se non si utilizzano questi simboli il programma cercherà di scrivere il valore di una variabile o una costante che abbia quel nome.

Gli altri argomenti sono interessanti per la realizzazione di "impaginazioni" elaborate, utili all'introduzione del web design, mostrando come l'utilizzo degli editor testuali (in alternativa agli editor visuali) permetta il pieno controllo della grafica.

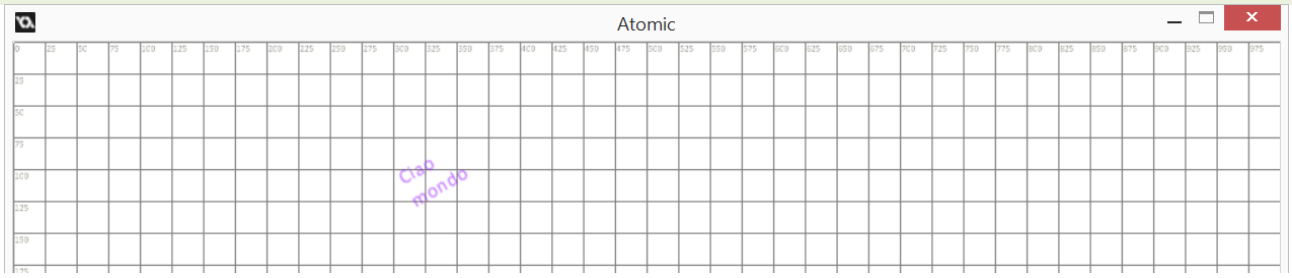
Esempio semplice per introdurre la funzione:

disegna testo --> (TESTO: "Ciao mondo") (X: 25) (Y: 25)



Esempio più complesso:

disegna testo --> (TESTO: "Ciao mondo") (X: 300) (Y: 100) (COLORE: viola) (TRASPARENZA: 0.65) (ROTAZIONE: 30)
(LARGHEZZA CASELLA: 50)



ottieni stile testo --> (CARATTERE:) (DIMENSIONI:) (GRASSSETTO:) (CORSIVO:)

La funzione ***ottieni stile testo*** permette di definire uno stile partendo da un carattere tipografico (vedi sezione COSTANTI, per la lista dei caratteri disponibili)

Esempio:

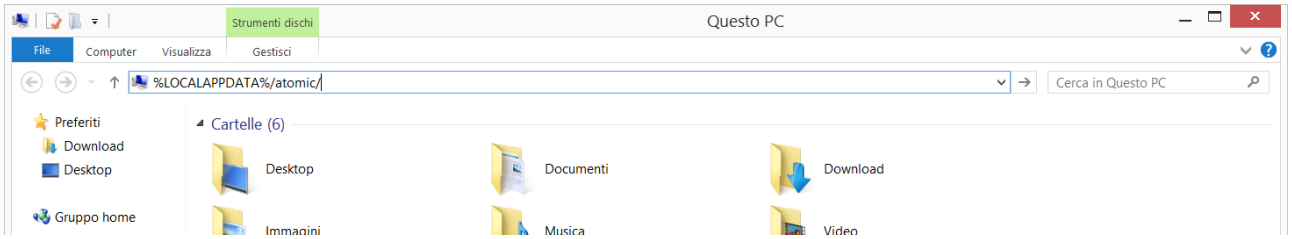
```
INIZIA
stile_fumetto = ottieni_stile_testo --> (CARATTERE: Comic Sans ) (DIMENSIONE: 100)
CICLO CONTINUO
disegna testo --> (TESTO: "Ciao mondo") (STILE: stile_fumetto ) (COLORE: blu )
```

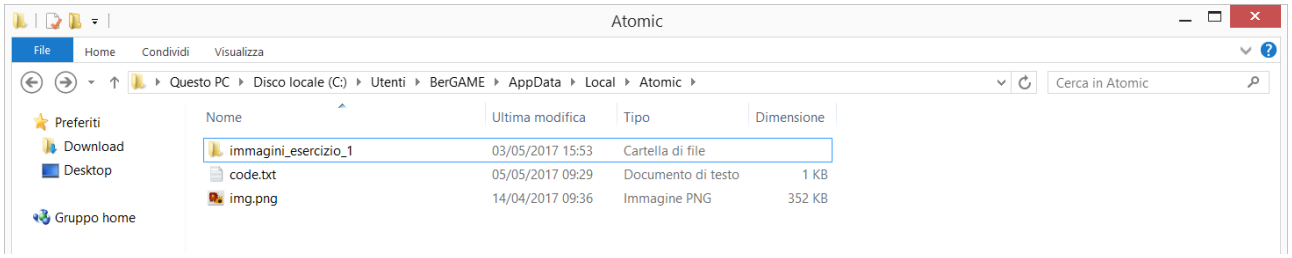
stile_fumetto nell'esempio è la variabile al cui interno viene memorizzato lo stile (è possibile usare qualsiasi nome valido come nome per la variabile). Il simbolo = che precede la funzione indica l'assegnazione di un dato alla variabile, in questo caso un dato complesso **ottenuto** da una funzione.


Atomic
Ciao mondo

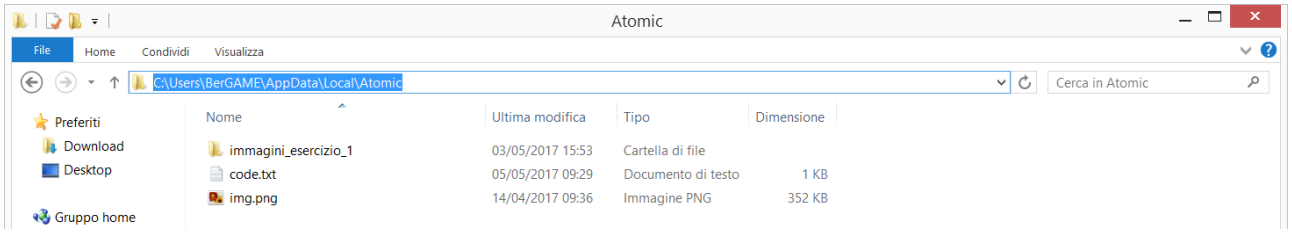
Funzioni di disegno delle immagini

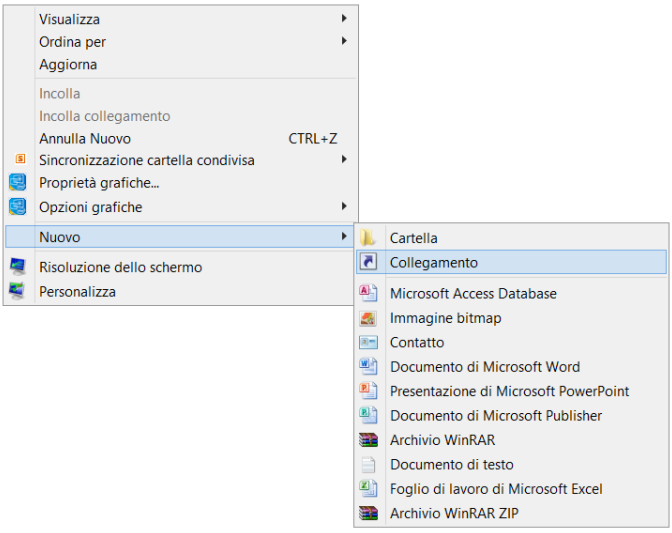
Con la funzione ***disegna immagine*** è possibile disegnare qualsiasi immagine di formato .png e .jpg.
Queste immagini per poter essere disegnate vanno copiate all'interno della cartella **%LOCALAPPDATA%/Atomic/** (copia e incolla l'indirizzo sulla barra di Explorer per accedere alla cartella).

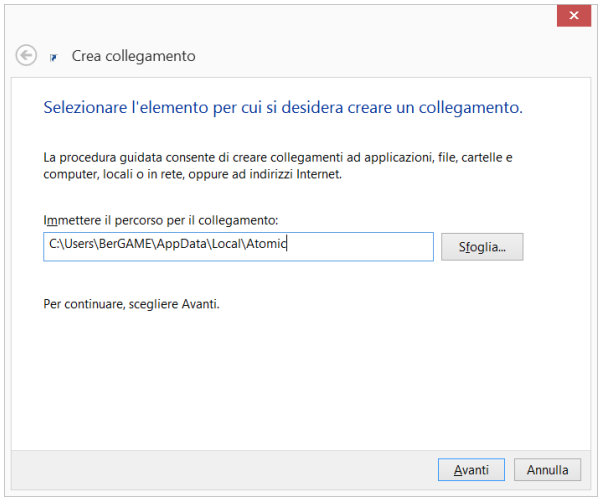




Una volta trovata la cartella è possibile visualizzare il vero indirizzo cliccando sulla barra (l'indirizzo varia a seconda del computer; nell'esempio è **C:\Users\BerGAME\AppData\Local\Atomic**); una volta visualizzato è possibile copiarlo e creare un comodo collegamento sul desktop (desktop -> click destro -> nuovo -> collegamento).







Una volta inserita un'immagine nella cartella è possibile utilizzarla in Atomic utilizzando la funzione *ottieni immagine:*

```
ottieni immagine --> (NOME: ) (ORIGINE X:) (ORIGINE Y:)
```

Esempio, dopo aver inserito l'immagine *fotografia_bella.jpg* nella cartella di Atomic:

```
foto = ottieni immagine --> (NOME: "fotografia_bella.jpg")
```

foto nell'esempio è la variabile al cui interno viene memorizzata l'immagine (è possibile usare qualsiasi nome valido come nome per la variabile). Il simbolo = che precede la funzione indica l'assegnazione di un dato alla variabile, in questo caso un dato complesso **ottenuto** da una funzione.

gli argomenti ORIGINE X e ORIGINE Y rappresentano il punto d'origine relativo da cui l'immagine viene disegnata. Normalmente è l'angolo in alto a sinistra ovvero x 0,y 0. Se per esempio la vostra immagine è larga 300 pixel e lunga 200 pixel e volete impostare l'origine al centro l'origine sarà x 150, y 100 .

Esempio:

```
foto = ottieni immagine --> (NOME: "fotografia_bella.jpg")  (ORIGINE X: 150 ) (ORIGINE Y: 100)
```

Impostare l'origine è utile quando si vuole giocare con la rotazione e la scala delle immagini.
Questa figura dovrebbe chiarire il concetto:

È anche possibile organizzare le immagini in cartelle, ad esempio se creo la cartella *immagini_gatti* dentro la cartella di Atomic e all'interno ci inserisco l'immagine *gatto.jpg*, posso richiamare l'immagine *gatto.jpg* in questo modo:

```
foto = ottieni immagine --> (NOME: "immagini_gatti/gatto.jpg")
```

Una volta ottenuta una variabile contenente l'immagine desiderata è possibile disegnarla usando la funzione *disegna immagine*:

```
disegna immagine -->  (IMMAGINE: ) (X:) (Y:) (SCALA ASSE X:) (SCALA ASSE Y:) (COLORE:) (TRASPARENZA:) (ROTAZIONE:)
```

Esempio:

```
foto = ottieni immagine --> (NOME: "fotografia_bella.jpg")
disegna immagine --> (IMMAGINE: foto) (X: 75) (Y: 75)
```

Se si inseriscono troppe immagini è possibile che non ci sia abbastanza spazio disponibile per memorizzarle tutte contemporaneamente (soprattutto se grandi). Se questo succede il programma si arresta (crash per “out of memory”).

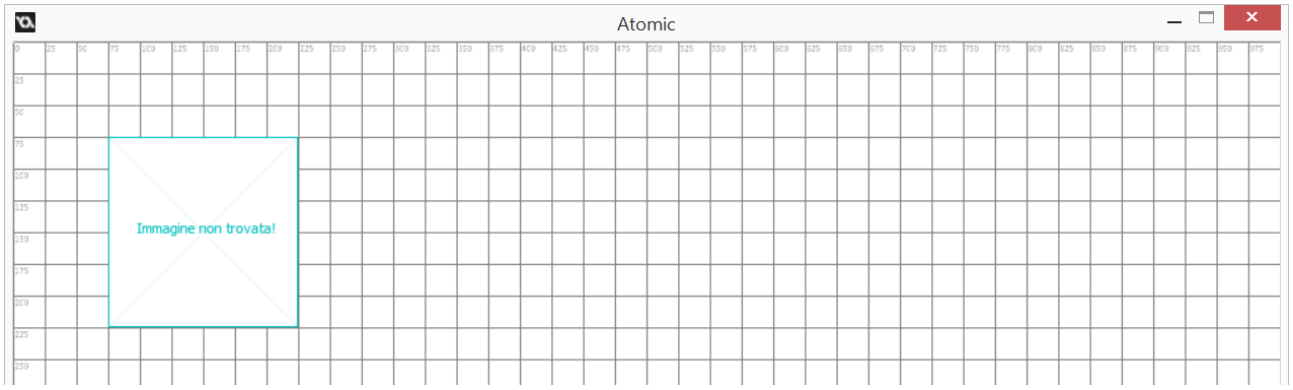
Per evitare ciò è possibile eliminare dalla memoria un’immagine che non ci serve più con la funzione *elimina immagine*.

```
elimina immagine --> (NOME: )
```

Esempio:

```
elimina immagine --> (NOME: foto)
```

Se un’immagine non esiste più o non è mai esistita (per esempio se si sbaglia a digitarne il nome) ma tentiamo comunque di disegnarla, al suo posto verrà visualizzata un’immagine alternativa con scritto “immagine non trovata!”.



È importante inserire le immagini che ci servono **una sola volta per ogni immagine;** il modo più facile per farlo è inserirle nell’evento INIZIA. Se si vuole inserire un’immagine nell’evento CICLO CONTINUO bisogna stare attenti che l’immagine venga inserita una sola volta altrimenti il programma terminerà in poco tempo visualizzando l’errore “out of memory” poiché ogni trentesimo di secondo verrà inserita un’immagine identica nella memoria!

Funzioni di casualità

ottieni uno a caso di questi valori --> (VALORE 1:) (VALORE 2:) (VALORE 3:) … (VALORE 8:)

ottieni un valore compreso tra questi --> (VALORE 1:) (VALORE 2:)

ottieni un valore intero compreso tra questi --> (VALORE 1:) (VALORE 2:)

attiva casualità diversa per ogni avvio

Queste funzioni permettono di assegnare valori casuali alle variabili. La funzione *attiva casualità diversa per ogni avvio* permette di ottenere una **casualità totale**: se questa funzione non viene utilizzata ogni volta che si avvia un determinato programma (sempre lo stesso, con nessuna modifica) la generazione casuale è sempre la stessa.

Esempio:

```
test = ottieni un valore intero compreso tra questi --> (VALORE 1: 10 ) (VALORE 2: 5)
```

supponiamo che con questa riga di codice la variabile *test* assuma il valore 8. Ogni volta che il programma verrà avviato il risultato sarà sempre 8.

Invece scrivendo:

```
attiva casualità diversa per ogni avvio
test = ottieni un valore intero compreso tra questi --> (VALORE 1: 10 ) (VALORE 2: 5)
```

Verosimilmente può succedere questo: la prima volta che il programma verrà avviato la variabile test assumerà il valore 8, la seconda volta 5, la terza 7, ecc..

Funzioni matematiche, trigonometriche e vettoriali

Queste funzioni sono utili per esercizi matematici e geometrici applicabili anche a materie come il disegno e la musica.

ottieni l'arrotondamento per difetto di --> (VALORE:)

ottieni l'arrotondamento per eccesso di --> (VALORE:)

ottieni il valore intero di --> (VALORE:)

ottieni il valore decimale di --> (VALORE:)

ottieni il valore assoluto di --> (VALORE:)

ottieni il segno di --> (VALORE:)

ottieni il massimo tra --> (VALORE 1:) (VALORE 2:) (VALORE 3:) … (VALORE 8:)

ottieni il minimo tra --> (VALORE 1:) (VALORE 2:) (VALORE 3:) … (VALORE 8:)

ottieni la media tra --> (VALORE 1:) (VALORE 2:) (VALORE 3:) … (VALORE 8:)

ottieni il valore più vicino alla media tra --> (VALORE 1:) (VALORE 2:) (VALORE 3:) … (VALORE 8:)

ottieni l'interpolazione lineare tra --> (VALORE 1:) (VALORE 2:) (PERCENTUALE:)

ottieni il valore limitato della variabile tra --> (VALORE 1:) (VALORE 2:) (VARIABILE:)

ottieni la funzione esponenziale di --> (VALORE:)

ottieni il logaritmo naturale di --> (VALORE:)

ottieni il logaritmo in base 2 di --> (VALORE:)

ottieni il logaritmo in base 10 di --> (VALORE:)

ottieni il logaritmo in base n di --> (VALORE:) (BASE:)

ottieni la potenza di --> (VALORE:) (ESPONENTE:)

ottieni il quadrato di --> (VALORE:)

ottieni la radice quadrata di --> (VALORE:)

ottieni il seno di --> (VALORE:) (UNITA:)

ottieni il coseno di --> (VALORE:) (UNITA:)

ottieni la tangente di --> (VALORE:) (UNITA:)

ottieni l'arcoseno di --> (VALORE:) (UNITA:)

ottieni l'arcocoseno di --> (VALORE:) (UNITA:)

ottieni l'arcotangente di --> (VALORE:) (UNITA:)

ottieni l’arcotangente 2 di --> (VALORE 1:) (VALORE 2:) (UNITA:)

L’argomento UNITA può essere la stringa “radianti” o la stringa “gradi”

ottieni il valore convertito da radianti a gradi di --> (VALORE:)

ottieni il valore convertito da gradi a radianti di --> (VALORE:)

ottieni la componente x del vettore dato angolo e lunghezza --> (LUNGHEZZA:) (ANGOLO:)

ottieni la componente y del vettore dato angolo e lunghezza --> (LUNGHEZZA:) (ANGOLO:)

ottieni distanza tra due punti --> (X1:) (Y1:) (X2:) (Y2:)

ottieni direzione tra due punti --> (X1:) (Y1:) (X2:) (Y2:)

ottieni differenza angolare tra--> (VALORE 1:) (VALORE 2:)

Funzioni sul testo (stringhe di testo)

Queste funzioni servono per manipolare stringhe di testo. La più utilizzata ed essenziale è sicuramente **ottieni testo combinato**.

ottieni testo combinato --> (TESTO 1:) (TESTO 2:) … (TESTO 8:)

Questa funzione permette di combinare fino ad otto stringhe assieme in un'unica stringa (somma di stringhe).
È utile nel caso si voglia creare del **testo contenente variabili**.

Esempio:

```
INIZIA
eta=10
scritta = ottieni testo combinato --> (TESTO 1: “Ciao! Io ho “) (TESTO 2: eta) (TESTO 3: “ anni!”)

CICLO CONTINUO
disegna testo --> (TESTO: scritta)
```

Questa funzione è l’alternativa alla somma di stringhe con l’operatore + (non supportata):

```
scritta = “Ciao! Io ho “+string(eta)+” anni!”;
```

Da notare che con *ottieni testo combinato* non è necessario trasformare un valore reale in stringa poiché la funzione lo fa automaticamente.

Altre funzioni per la manipolazione di stringhe di testo:

ottieni numero da testo --> (TESTO:)

ottieni il simbolo di un testo a una determinata posizione --> (TESTO:) (POSIZIONE:)

ottieni il numero di parole specificate in un testo --> (TESTO:) (PAROLA:)

ottieni lunghezza testo --> (TESTO:)

ottieni testo con parola sostituita --> (TESTO:) (PAROLA:) (NUOVA PAROLA:)

ottieni testo con parola sostituita solo la prima volta --> (TESTO:) (PAROLA:) (NUOVA PAROLA:)

Funzioni sul colore

Queste funzioni sono utili per creare colori personalizzati e ad introdurre i sistemi di colore RGB e HSV.

ottieni colore rgb --> (ROSSO:) (VERDE:) (BLU:)

ottieni colore hsv --> (TINTA:) (SATURAZIONE:) (LUMINOSITA:)
Gli argomenti devono essere valori interi compresi tra 0 e 255

ottieni miscela colori da --> (COLORE 1:) (COLORE 2:) (VALORE:)
L'argomento VALORE deve essere compreso tra 0 e 1: 0 = COLORE 1, 1 = COLORE 2, 0.5 = miscela al 50% tra i due colori.

Funzioni sull'audio

La funzione principale per quanto riguarda l'audio è *suona* che permette di emettere un suono impostandone anche il volume (inteso come *gain*) e l'intonazione.

```
suona --> (SUONO:) (VOLUME:) (INTONAZIONE:)
```

è possibile utilizzare qualsiasi numero intero per l'intonazione ma è consigliabile utilizzare le **note musicali** (vedi sezione costanti).

Altre funzioni sull'audio sono:

```
imposta volume principale --> (VOLUME:)
```
1=massimo, 0=muto

```
ottieni volume principale
```

```
ottieni valore volume di --> (SUONO:)
```

```
ottieni valore intonazione di -->  (SUONO:)
```

Tasti virtuali

I tasti virtuali sono degli **oggetti** speciali disegnati all'interno della finestra. Questi tasti possono avere caratteristiche grafiche personalizzate (colore, testo, dimensioni…) e rendono possibile programmare delle azioni tramite l'utilizzo del mouse: ad esempio emettere un suono, disegnare una forma o svolgere qualsiasi altra funzione o gruppo di funzioni.

Per creare un tasto virtuale si utilizza la funzione *crea tasto virtuale*:

```
crea tasto virtuale --> (NOME: ) (ETICHETTA: ) (X: ) (Y: ) (COLORE SFONDO: ) (COLORE TESTO: ) (TRASPARENZA: ) (LARGHEZZA: )
                        (ALTEZZA: ) (CLASSE:)
```

L'argomento ETICHETTA è una stringa di testo che viene disegnata all'interno del tasto, questa può essere diversa dal nome.
L'argomento CLASSE è utile per definire la grafica di tasti simili usando un solo argomento (vedi poco più avanti).

Per poter utilizzare un tasto virtuale è necessario utilizzare la funzione *ottieni stato tasto virtuale* per sapere se: *è cliccato*, *è stato cliccato* o *non è cliccato*:

```
ottieni stato tasto virtuale --> (NOME:)
```

La variabile memorizza quindi uno di questi valori:
è cliccato = l'utente sta premendo il tasto (pressione continua)
non è cliccato = l'utente non sta premendo il tasto
è stato cliccato = l'utente ha premuto il tasto (singolo click)

Esempio:

```
INIZIA
crea tasto virtuale --> (NOME: tasto_di_prova) (ETICHETTA: "Cliccami!")

CICLO CONTINUO
tasto1 = ottieni stato tasto virtuale --> (NOME: tasto_di_prova)
se tasto1 è cliccato allora disegna cerchio .
```

Questo esempio disegna un cerchio se il tasto viene cliccato.

Quando bisogna creare molti tasti simili può essere utile creare una **classe** di tasti per **definire una sola volta** il loro aspetto e la loro posizione. Questa funzione rende molto più veloce la scrittura del codice e la sua modifica, inoltre lo rende più snello e leggibile.

```
crea classe di tasti --> (NOME:) (ETICHETTA: ) (X: ) (Y: ) (COLORE SFONDO: ) (COLORE TESTO: ) (TRASPARENZA: ) (LARGHEZZA: )
                         (ALTEZZA:)
```

Esempio:

```
INIZA
crea classe di tasti --> (NOME: classe_di_prova) (X: 600) (COLORE SFONDO: nero) (COLORE TESTO: giallo) (ALTEZZA: 50)
                          (ETICHETTA: "sono un tasto!") (TRASPARENZA: 0.75)
crea tasto virtuale --> (NOME: tasto_a) (CLASSE: classe_di_prova) (Y: 50)
crea tasto virtuale --> (NOME: tasto_b) (CLASSE: classe_di_prova) (Y: 150)
crea tasto virtuale --> (NOME: tasto_c) (CLASSE: classe_di_prova) (Y: 250) (ETICHETTA: "")
crea tasto virtuale --> (NOME: tasto_d) (CLASSE: classe_di_prova) (Y: 350)
```

Quando un tasto virtuale non serve più possiamo eliminarlo con la funzione *distruggi tasto virtuale.*

```
distruggi tasto virtuale --> (NOME:)
```

Esempio:

```
distruggi tasto virtuale --> (NOME: tasto_a)
```

Funzioni miscellanea

Queste sono le funzioni che attualmente non rientrano in nessuna categoria:

imposta griglia --> (VISIBILE:) (COLORE SFONDO:) (COLORE LINEE:) (DIMENSIONE:)

salva schermata

esci dal programma

riavvia il programma

Funzioni previste per le prossime versioni:

- Funzioni aggiuntive riguardanti l'interazione utente (menu, domande, richieste all'utente)
- Funzioni per la gestione di oggetti e classi di oggetti
- Funzioni per la comunicazione con Arduino
- Funzioni di disegno aggiuntive
- Funzioni per inserire suoni e musica personalizzati
- Funzioni aggiuntive sulle tabelle (vedi sezione TABELLE)
- Altre funzioni suggerite dagli utenti

AUMENTA E DIMINUISCI

In ATOMIC esistono dei costrutti per aumentare e diminuire in modo intuitivo il valore di una variabile

Per aumentare di 1 la variabile *orsetto_lavatore* si può scrivere:

```
orsetto_lavatore = orsetto_lavatore +1
```

Ovvero prende il valore di se stesso e somma 1

In altri linguaggi questa operazione può essere scritta in questo modo:

```
orsetto_lavatore +=1;
```

o addirittura

```
orsetto_lavatore ++;
```

Queste scritture sono sicuramente veloci e comode per un programmatore navigato ma molto distanti dal linguaggio umano.

In ATOMIC è possibile scrivere l'operazione soprariportata in questo modo:

```
aumenta orsetto_lavatore di 1
```

o per diminuirla, per esempio di 2:

```
diminusici orsetto_lavatore di 2
```

è anche possibile diminuire o aumentare in base al risultato di una espressione, ad esempio scrivendo:

```
aumenta orsetto_lavatore di 5*2
aumenta orsetto_lavatore di gatto
diminuisci topo_ragno  di 1+2*(6/orsetto_lavatore)
```

COMMENTI

Come la maggior parte dei linguaggi di programmazione ATOMIC supporta i commenti.
I commenti sono parti del codice che vengono ignorate dal computer.

È possibile commentare una linea di codice utilizzando il simbolo "//". Esempio:

```
// questo è un commento, la funzione qui sotto disegna un cerchio con 200 di raggio alla posizione 300x 400y
disegna cerchio --> (RAGGIO: 200) (X: 300) (Y: 400)
//disegna cerchio --> (RAGGIO: 100) (X: 500) (Y: 600)
```

La prima e la terza linea verranno ignorate dal computer.

Lo scopo dei commenti è duplice.
In primo luogo permette di inserire all'interno del codice delle descrizioni che migliorano la comprensione da parte degli umani, facilitandone la condivisione e il lavoro a lungo termine.
In seconda istanza permette di eliminare temporaneamente parti di codice senza doverle cancellare; in pratica permettono di disattivare pezzi di codice.

Per quest'ultimo utilizzo vengono comodi i commenti multilinea, che utilizzano i simboli "/*" e "*/". Esempio:

```
/* tutto questo pezzo di codice verrà ignorato
disegna cerchio --> (RAGGIO: 200) (X: 300) (Y: 400)
disegna cerchio --> (RAGGIO: 100) (X: 500) (Y: 600) */
```

COSTANTI

Le costanti sono valori non modificabili già integrati in ATOMIC.
Possono essere utilizzate nello stesso modo delle variabili.

Lista delle costanti matematiche:

NOME	VALORE
vero, intero, intera	1
falso, niente, nullo	0
pi greco	3,1415926535 …
numero di nepero	2,7182818284 ...
sezione aurea	1,6180339887...
mezza, mezzo, un mezzo	0,5
un terzo	0,333333…
un quarto	0,25
un quinto	0,2
un sesto	0,16666666…
un settimo	0,1428571…
un ottavo	0,125
un nono	0,1111111111…
un decimo	0,1
tre quarti	0,75
due terzi	0,666666666…

Costanti di semplificazione scrittura del codice

N.B. queste costanti non sono utilizzabili nelle espressioni!

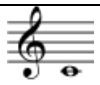
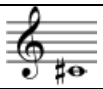
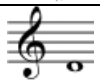
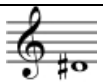
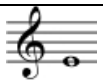
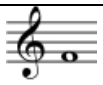
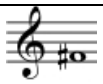
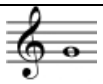
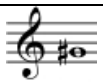
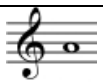
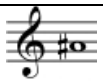
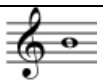
NOME	VALORE
è stato cliccato	“è stato cliccato”
non è cliccato	“non è cliccato”
è cliccato	“è cliccato”
al centro	“al centro”
a destra	“a destra”
a sinistra	“a sinistra”
in alto	“in alto”
in basso	“in basso”
radianti	“radianti”
gradi	“gradi”
è uguale a	“=”
è maggiore di	“>”
è minore di	“<”
è maggiore o uguale a	“>=”
è minore o uguale a	“<=”
è diverso da	“!=”

Lista delle costanti che rappresentano un colore:

NOME	COLORE	VALORE RGB
arancione		255,160,64
argento		220,220,220
azzurro		0,127,255
bianco		255,255,255
blu		0,0,255
bronzo		205,127,50
castagno		205,92,92
castagno scuro		152,105,96
ciano		0,255,255
corallo		255,127,80
giallo		255,255,0
grigio		192,192,192
grigio chiaro		242,242,242
grigio scuro		129,128,128

magenta		255,0,255
marrone		128,0,0
nero		0,0,0
oro		255,192,0
rosa		255,192,203
rosso		255,0,0
rosso vermiglione		227,66,52
verde		0,255,0
verde acqua		0,128,128
verde oliva		128,128,0
verde scuro		0,128,0
viola		143,0,255

Costanti musicali

NOME	NOTA	VALORE
Do		1
Do diesis Re bemolle		1.061068702290076
Re		1.122137404580153
Re diesis Mi bemolle		1.190839694656489
Mi		1.259541984732824
Fa		1.33587786259542
Fa diesis Sol bemolle		1.412213740458015
Sol		1.49618320610687
Sol diesis La bemolle		1.587786259541985
La		1.679389312977099
La diesis Si bemolle		1.778625954198473
Si		1.885496183206107

Le costanti musicali possono essere utilizzate per intonare un suono.

Esempio:

suona --> (SUONO: suono miao) (INTONAZIONE: Fa diesis)

Di base un suono è relativamente intonato in Do.
Questo perché la frequenza “normale” del miagolio di un gatto è diversa ad esempio da quella del muggito di una mucca.
Per intonare due suoni tra loro su una stessa nota bisogna applicare una correzione a uno dei due suoni sommando o sottraendo una nota (aumentando o diminuendo la frequenza).

Esempio:

```
INIZIA
correzione = La

CICLO CONTINUO
suona --> (SUONO: suono muu) (INTONAZIONE: correzione + Fa diesis)
suona --> (SUONO: suono miao) (INTONAZIONE: Fa diesis)
```

Per utilizzare note di ottave superiori o inferiori basta moltiplicare o dividere la nota per due elevato al numero di ottava desiderato. Ovvero ***Nota*2^Ottava*** o ***Nota/2^Ottava***.

Tre ottave inferiori	La/2^3	La/8
Due ottave inferiori	La/2^2	La/4
Una ottava inferiore	La/2^1	La/2
Normale (ipotetica La4)	La*2^0	La
Una ottava superiore	La*2^1	La*2
Due ottave superiori	La*2^2	La*4
Tre ottave superiori	La*2^3	La*8

Questa formula in teoria è sempre valida: ipoteticamente se volessimo ottenere 20 ottave superiori potremmo scrivere La*2^20 ovvero La*1048576, tuttavia il suono ottenuto sarebbe talmente alto da risultare impercepibile da un umano. Per questo stesso motivo a volte anche lavorando con ottave "ragionevoli" il suono potrebbe non sentirsi se la sua intonazione di base è molto bassa o molto alta. L'ideale è lavorare con suoni intonati di base in Do4 ovvero 262Hz.

Esempio:

```
suona --> (SUONO: suono beep1) (INTONAZIONE: DO*2) //intonato in DO di una ottava più alta
suona --> (SUONO: suono beep2) (INTONAZIONE: FA/4) //intonato in FA di due ottave più basse
```

Alcune costanti indicano una ***risorsa*** come un *suono* o un *font* integrato in ATOMIC.

Lista delle costanti che rappresentano un suono:

NOME	DESCRIZONE
suono beep1	Suono di interfaccia
suono beep2	Suono di interfaccia
suono beep3	Suono di interfaccia
suono miao	Verso del gatto
suono bau	Verso del cane
suono quak	Verso dell'oca
suono beee	Verso della pecora
suono squit	Verso del topo
suono muuu	Verso della mucca
suono uhahah	Verso della scimmia
suono hiii	Verso del cavallo
suono hiho	Verso dell'asino
suono driiin	Suono del campanello
suono errore	Suono negativo di interfaccia
suono ok	Suono positivo di interfaccia

Lista delle costanti che rappresentano un font (carattere tipografico):

Nome	Esempio
Calibri	Cantami o diva del pelide Achille l'ira funesta che infinite addusse 0 1 2 3 4 5 6 7 8 9
Verdana	Cantami o diva del pelide Achille l'ira funesta che infinite addusse 0 1 2 3 4 5 6 7 8 9
Times New Roman	Cantami o diva del pelide Achille l'ira funesta che infinite addusse 0 1 2 3 4 5 6 7 8 9
Tahoma	Cantami o diva del pelide Achille l'ira funesta che infinite addusse 0 1 2 3 4 5 6 7 8 9
Comic Sans	Cantami o diva del pelide Achille l'ira funesta che infinite addusse 0 1 2 3 4 5 6 7 8 9
Brush Script	*Cantami o diva del pelide Achille l'ira funesta che infinite addusse 0 1 2 3 4 5 6 7 8 9*
Old English Text	Cantami o diva del pelide Achille l'ira funesta che infinite addusse 0 1 2 3 4 5 6 7 8 9
Chiller	Cantami o diva del pelide Achille l'ira funesta che infinite addusse 0 1 2 3 4 5 6 7 8 9
Gigi	Cantami o diva del pelide Achille l'ira funesta che infinite addusse 0 1 2 3 4 5 6 7 8 9
Bauhaus 93	**Cantami o diva del pelide Achille l'ira funesta che infinite addusse 0 1 2 3 4 5 6 7 8 9**

COSTRUTTO SE

Il costrutto ***se*** permette di eseguire un pezzo di codice solo se una certa condizione è *vera*.
La sintassi da utilizzare è la seguente:

se <espressione> <confronto> <espressione> allora <esegui questo codice> .

Esempio:

se gatto = 1 allora disegna cerchio --> (RAGGIO: 200) (COLORE: blu) .

se la variabile gatto è uguale a 1 tutto il codice compreso tra "**allora**" e il simbolo "**.**" viene eseguito, altrimenti viene ignorato.

è possibile utilizzare i seguenti operatori:

OPERATORE	DESCRIZIONE	SIMBOLO EQUIVALENTE IN MATEMATICA
=	è uguale a	=
>	è maggiore di	>
<	è minore di	<
<=	è maggiore o uguale a	≥
>=	è minore o uguale a	≤
!=	è diverso da	≠

Esempi:

se gatto > 0 allora disegna cerchio --> (RAGGIO: 200) (COLORE: blu) .
se cane < 10 allora disegna cerchio --> (RAGGIO: 100) (COLORE: rosso) .
se topo != cane allora disegna cerchio --> (RAGGIO: 150) (COLORE: giallo) .
se cane/pi greco <= (10-2)*gatto allora disegna cerchio --> (RAGGIO: 70) (COLORE: arancione) .

è possibile utilizzare la forma estesa (descrizione) al posto del simbolo. Esempio:

se gatto è maggiore di 0 allora disegna cerchio --> (RAGGIO: 200) (COLORE: blu) .
se cane è minore di 10 allora disegna cerchio --> (RAGGIO: 100) (COLORE: rosso) .
se topo è diverso da cane allora disegna cerchio --> (RAGGIO: 150) (COLORE: giallo) .
se cane/pi greco è minore o uguale a (10-2)*gatto allora disegna cerchio --> (RAGGIO: 70) (COLORE: arancione) .

è anche possibile inserire più righe di codice controllate da un *se*. Esempio:

se cane = 1 allora
disegna cerchio --> (RAGGIO: 100) (COLORE: rosso)
disegna rettangolo --> (LARGHEZZA: 200) (COLORE: verde)
topo = 300
.

In questi casi può essere più comodo, come in altri linguaggi di programmazione, utilizzare le **parentesi graffe "{ }"** al posto di "allora" e "." In modo da identificare a vista d'occhio i blocchi di codice.
Parentesi graffa aperta: { = allora
Parentesi graffa chiusa: } = .

Esempio, equivalente a quello riportato sopra:

se cane = 1
{
disegna cerchio --> (RAGGIO: 100) (COLORE: rosso)
disegna rettangolo --> (LARGHEZZA: 200) (COLORE: verde)
topo = 300
}

Nulla vi vieta di usare questa notazione anche per delle singole linee se lo ritenete più comodo. Esempio:

se gatto > 0 { disegna cerchio --> (RAGGIO: 200) (COLORE: blu) }

è anche possibile inserire dei controlli *se* dentro altri controlli *se*. Esempio:

INIZIA
gatto=0

```
cane=1

CICLO CONTINUO
se gatto = 0
allora
     se cane < 101
     allora
     disegna cerchio --> (RAGGIO: cane)
     se cane < 100 allora aumenta cane di 1 .
     .
.
```

Che può anche essere scritto in questo modo:

```
INIZIA
gatto=0
cane=1

CICLO CONTINUO
se gatto = 0
{
   se cane <= 100
   {
   disegna cerchio --> (RAGGIO: cane)
   se cane < 100  {aumenta cane di 1 }
   }
}
```

Questo esempio disegna un cerchio che partendo da un raggio di 1 pixel cresce fino ad avere un raggio di 100 pixel.

È altresì possibile fare confronti di uguaglianza con stringhe di testo, ad esempio:

```
INIZIA
gatto="persiano"
cane="carlino"

CICLO CONTINUO
se gatto = "certosino"
{
se cane = "carlino" allora disegna cerchio --> (RAGGIO: 100) .
}
```

VARIANTE ABBREVIATA DEL COSTRUTTO SE

Esiste anche una variante abbreviata del costrutto se:

```
se   <espressione>   allora    <esegui questo codice>   .
```

ad esempio questo pezzo di codice:

```
se tasto invio è stato premuto = vero allora suona --> ( SUONO : suono bau ) .
//fai abbaiare il computer quando si preme il tasto invio
```

può essere abbreviato in questo modo, rendendo il codice più leggibile:

```
se tasto invio è stato premuto allora suona --> ( SUONO : suono bau ) .
//fai abbaiare il computer quando si preme il tasto invio
```

Non scrivendo l'operatore di confronto e l'espressione da confrontare il computer da per scontato che vogliamo controllare che il risultato dell'espressione (nell'esempio sopra una singola variabile) sia vero.
Questa forma è utile da utilizzare con le variabili che contengono valori binari (1-0 ovvero vero-falso).

OPERATORI LOGICI

ATOMIC non supporta le espressioni booleane ma supporta alcuni basilari operatori logici che si possono utilizzare all'interno del costrutto se. Il supporto agli operatori logici attualmente è limitato ma non è escluso che venga inserito un vero supporto alle espressioni logiche in futuro. Il motivo è che l'uso estensivo degli operatori logici permette di abbreviare molto il codice, rendendo potenzialmente criptici molti passaggi fondamentali per la comprensione dell'algoritmo o comunque può renderli intrinsecamente macchinosi da comprendere, poiché semanticamente poco chiari. È ancora da studiare un sistema alternativo alle espressioni booleane che renda più chiara e schematica la comprensione, tramite funzioni e/o costrutti speciali.

Per ottenere l'operatore **NOT** ("!"), che a differenza degli altri è unario è possibile usare questa funzione:

```
ottieni la negazione di --> (VALORE:)
```

Semplicemente NOT inverte il vero con il falso e viceversa.

Esempio:

```
gatto=vero
cane=0
gatto2 = ottieni negazione di --> (VALORE: gatto)
cane2 = ottieni negazione di --> (VALORE: cane)
```

Come già specificato nella sezione "costanti" ricordo che 1=vero e 0=falso.
gatto2 è falso, cane2 è 1 (vero).

È anche possibile invertire il valore della variabile negando la variabile stessa, esempio:

```
INIZIA
luce_accesa=0

CICLO CONTINUO
se tasto invio è stato premuto allora luce_accesa = ottieni negazione di --> (VALORE: luce_accesa) .
```

Questo esempio spegne/accende una luce premendo il tasto invio.
Questa logica è riassumibile in queste due frasi:
se è vero che la luce è accesa allora fai in modo che **non** sia vero che la luce è accesa (se la luce è accesa spegnila).
Se è falso che la luce è accesa allora fai in modo che **non** sia falso che la luce è accesa (se la luce è spenta accendila).

Questi sono gli operatori logici binari attualmente supportati:

NOME	VERO NOME	"SIMBOLO INFORMATICO"	SIMBOLO MATEMATICO
e	AND	&&	∧
o	OR	||	∨
o esclusivamente	XOR	^^	⊕

A differenza di altri linguaggi **è possibile combinare un solo operatore binario per ogni costrutto *se***.

Esempi:

```
se coniglio= 1 e anatra= 0 allora gatto=1 .
se anatra= 1 o pollo= 0 allora gatto=2 .
se tacchino = 1 o esclusivamente pollo= 0 allora gatto=3 .
```

"**e**" restituisce vero se entrambe le espressioni sono vere → $(x \wedge y)=1$ solo se $(x=1 \wedge y=1)$
"**o**" restituisce vero se almeno una delle espressioni è vera → $(x \vee y)=1$ solo se $(x=0 \wedge y=1) \vee (x=1 \wedge y=0) \vee (x=1 \wedge y=1)$
"**o esclusivamente**" restituisce vero solo se una delle due espressioni è vera e l'altra falsa → $(x \oplus y)=1$ solo se $(x=0 \wedge y=1) \oplus (x=1 \wedge y=0)$

Gli operatori logici non sono utilizzabili nella forma abbreviata.

In realtà anche se il supporto è limitato utilizzando alcune tecniche è possibile riprodurre il funzionamento della maggior parte delle espressioni logiche.

Prendiamo ad esempio questo codice (che ha un senso ma non è supportato da Atomic):

```
se (coniglio= 1 e anatra= 0) o esclusivamente (cane=1 o (pollo=1 e tacchino=0)) allora gatto=vero .
```

Può essere riprodotto scomponendolo in questo modo:

```
c1=0 //variabile contatore verità primo pezzo (coniglio, anatra)
c2=0 //variabile contatore verità secondo pezzo (cane, pollo, tacchino)

//c1
se coniglio=vero e anatra=falso allora c1=vero .

//c2
se cane=vero allora c2=vero .
se pollo=vero e tacchino=falso  allora  c2=vero .

//controllo finale di c1 e c2
se c1=vero o esclusivamente c2=vero allora gatto = vero .
```

Scritto in questo modo il codice è più ingombrante ma molto più schematico e chiaro.

Tutti gli operatori logici possono essere scomposti in semplici istruzioni se, qui sotto verranno illustrate le tecniche per farlo. Queste tecniche possono anche essere combinate tra loro per simulare espressioni booleane complesse. Potete anche accorciare il codice utilizzando gli operatori logici nei passaggi in cui sono semanticamente facili da comprendere, ricordando comunque che l'obbiettivo è schematizzare il ragionamento scomponendolo in piccole proposizioni logiche.

L'operatore AND ("e") è il meno problematico da scomporre e può essere simulato semplicemente indentando una seria di *se*.

```
se alfa=1 e (beta=2 e (gamma=23 e delta=74)) allora omega = 100 .
```

Come se ci trovassimo davanti ad una serie di addizioni le parentesi vanno bellamente ignorate; l'esempio diventa:

```
se alfa=1
{
  se beta=2
  {
    se gamma=23
    {
       se delta=74 allora omega = 100 .
    }
  }
}
```

L'operatore OR ("o") può essere traslato come una lista (se c'è almeno una cosa vera nella lista allora la condizione è vera)

```
se alfa=1 o (beta=2 o (gamma=23 o delta=74)) allora omega = 100 .
```

anche in questo caso le parentesi non hanno importanza e possono essere tralasciate:

```
se alfa=1 allora omega = 100 .
se beta=2 allora omega = 100 .
se gamma=23 allora omega = 100 .
se delta=74 allora omega = 100 .
```

L'operatore XOR ("o esclusivamente") è il più complesso da scomporre e richiede l'utilizzo di una variabile contatore ("i").

```
se alfa=1 o esclusivamente (beta=2 o esclusivamente (gamma=23 o esclusivamente delta=74)) allora omega = 100 .
```

Diventa:

```
i=0
se alfa=1 allora aumenta i di 1 .
se beta=2 allora aumenta i di 1 .
se gamma=23 allora aumenta i di 1 .
se delta=74 allora aumenta i di 1 .
```

```
se i=1
{
se alfa=1 allora omega = 100 .
se beta=2 allora omega = 100 .
se gamma=23 allora omega = 100 .
se delta=74 allora omega = 100 .
}
```

Come potete notare anche in questo caso le parentesi non hanno importanza (non hanno mai rilevanza in una espressione logica contenente un solo tipo di operatore). L'utilizzo della *variabile i* è necessario per verificare quante espressioni sono vere. A differenza di OR ("o"), la condizione per cui XOR restituisca vero è che solo una delle proposizioni sia vera, se due o più sono vere la condizione non è soddisfatta.

Questo è un esempio di scomposizione con più tipi di operatori logici:

```
se alfa=1 e (beta=2 o (gamma=23 o esclusivamente delta=74)) allora omega = 100 .
```

Una scomposizione corretta è questa:

```
se alfa=1
{
i=0
se gamma=23 allora aumenta i di 1 .
se delta=74 allora aumenta i di 1.
se beta=2 allora i=1 .
se i=1 allora omega = 100 .
}
```

In questo caso le parentesi sono rilevanti. Non esiste un metodo univoco per scomporre un'espressione logica in proposizioni meno complesse, l'importante è che funzioni e che il metodo sia chiaro e attinente al conteso.

COSTRUTTO FINCHE

Il costrutto ***finche*** ha una sintassi molto simile al costrutto *se*:
finche <espressione> <confronto> <espressione> allora <esegui questo codice> .

Esempio:

```
finche gatto < 100 allora aumenta gatto di 1 .
```

questo pezzo di codice aumenta **istantaneamente** la variabile gatto, portandola a 100.
Questo perché il costrutto *finche* esegue un **ciclo**, ovvero svolge una **iterazione**: ripete l'operazione alla massima velocità di calcolo possibile finche non ottiene il risultato per cui lavora; nel nostro caso finche la variabile gatto non sarà minore di 100, quindi quando gatto sarà maggiore o uguale a 100.

Qual è lo scopo?
Se il nostro obbiettivo è portare istantaneamente la variabile gatto a 100 basta usare una semplice assegnazione:

```
gatto = 100
```

perché allora utilizzare un ciclo finche?
La risposta è: per potere fare più cose contemporaneamente utilizzando una sola variabile **mentre** (while) raggiunge il valore 100, senza dover scrivere centinaia di righe di codice simili.

Un esempio pratico
Mettiamo il caso che vogliamo disegnare 20 cerchi equidistanti; possiamo scrivere:

```
INIZIA

CICLO CONTINUO
disegna cerchio --> (RAGGIO: 30) (COLORE: rosso) (X: 40)
disegna cerchio --> (RAGGIO: 30) (COLORE: rosso) (X: 40*2)
disegna cerchio --> (RAGGIO: 30) (COLORE: rosso) (X: 40*3)
disegna cerchio --> (RAGGIO: 30) (COLORE: rosso) (X: 40*4)
disegna cerchio --> (RAGGIO: 30) (COLORE: rosso) (X: 40*5)
disegna cerchio --> (RAGGIO: 30) (COLORE: rosso) (X: 40*6)
disegna cerchio --> (RAGGIO: 30) (COLORE: rosso) (X: 40*7)
disegna cerchio --> (RAGGIO: 30) (COLORE: rosso) (X: 40*8)
disegna cerchio --> (RAGGIO: 30) (COLORE: rosso) (X: 40*9)
disegna cerchio --> (RAGGIO: 30) (COLORE: rosso) (X: 40*10)
disegna cerchio --> (RAGGIO: 30) (COLORE: rosso) (X: 40*11)
disegna cerchio --> (RAGGIO: 30) (COLORE: rosso) (X: 40*12)
disegna cerchio --> (RAGGIO: 30) (COLORE: rosso) (X: 40*13)
disegna cerchio --> (RAGGIO: 30) (COLORE: rosso) (X: 40*14)
disegna cerchio --> (RAGGIO: 30) (COLORE: rosso) (X: 40*15)
disegna cerchio --> (RAGGIO: 30) (COLORE: rosso) (X: 40*16)
disegna cerchio --> (RAGGIO: 30) (COLORE: rosso) (X: 40*17)
disegna cerchio --> (RAGGIO: 30) (COLORE: rosso) (X: 40*18)
disegna cerchio --> (RAGGIO: 30) (COLORE: rosso) (X: 40*19)
disegna cerchio --> (RAGGIO: 30) (COLORE: rosso) (X: 40*20)
```

Questo codice è corretto ma è lungo e ridondante.

La stessa cosa utilizzando *finche* si può scrivere in questo modo, utilizzando la variabile *cerchi* come unità immaginaria:

```
INIZIA
cerchi=0

CICLO CONTINUO
cerchi=1
finche cerchi < 20 allora disegna cerchio --> (RAGGIO: 30) (COLORE: rosso) (X: 40*cerchi)
aumenta cerchi di 1 .
```

Il funzionamento è il seguente:

- ogni trentesimo di secondo la variabile *cerchi* viene riportata al valore 1 prima che inizi il ciclo
- ad ogni passaggio del ciclo viene disegnato un cerchio alla posizione x 40 per il numero di cerchi già disegnati
- ad ogni passaggio del ciclo la variabile *cerchi* viene aumentata di 1, il ciclo avrà quindi 20 passaggi (finche cerchi < 20)

In realtà è possibile fare molto di più, ad esempio modificare il raggio dei cerchi in modo progressivo:

```
INIZIA
cerchi=0

CICLO CONTINUO
cerchi=1
finche cerchi < 20 allora disegna cerchio --> (RAGGIO: 5*cerchi) (COLORE: rosso) (X: 40*cerchi)
aumenta cerchi di 1 .
```

Il costrutto *finche* è lo strumento più potente presente in ATOMIC, le sue applicazioni sono infinite e permettono di automatizzare una valanga di compiti noiosi e ripetitivi.

--- ATTENZIONE A NON CREARE CICLI INFINITI! ---

I cicli *finche* eseguono il codice finche la condizione è vera, se questa condizione è sempre vera il ciclo non terminerà mai e il programma si bloccherà irrimediabilmente (**crash**).

Ad esempio, questo codice farà sicuramente crashare il programma (fidatevi, non provatelo!):

```
INIZIA
prova=150

CICLO CONTINUO
finche prova > 100 allora disegna cerchio --> (RAGGIO: 5*prova) (COLORE: rosso) (X: 40*prova)
aumenta prova di 1 .
```

la variabile prova è sempre maggiore di 100, quindi il computer dovrebbe disegnare un numero infinito di cerchi:
il programma si blocca in attesa che il computer finisca di calcolare dove disegnare tutti i cerchi… Ma il calcolo non finirà mai!

COSTRUTTO RIPETI PER

Il costrutto *ripeti per* è una forma semplificata del costrutto *finche* e in molti casi può sostituirlo.
La sintassi è la seguente:
ripeti per <espressione> volte: <esegui questo codice> .

Esempio:

```
INIZIA
cerchi=0

CICLO CONTINUO
cerchi=1
ripeti per 20 volte : disegna cerchio --> (RAGGIO: 5*cerchi) (COLORE: rosso) (X: 40*cerchi)
aumenta cerchi di 1 .
```

La differenza con il ciclo finche è che il numero di ripetizioni è già definito in partenza.

Come nella maggior parte dei linguaggi di programmazione il ciclo *ripeti per* (*for*) e il ciclo *finche* (*while*) possono essere utilizzati per svolgere gli stessi compiti in base alla comodità del programmatore.
Anche per il ciclo *ripeti per* è possibile utilizzare le parentesi graffe al posto dei simboli “:” e “.”.

TABELLE (ARRAYS)

Attenzione: alcune funzioni sulle tabelle non sono ancora state corrette o implementate!

In ATOMIC gli arrays sono chiamati **tabelle**.
Le tabelle sono strutture di dati, dei contenitori per memorizzare, ed in seguito accedere, a grandi quantità di informazioni.
Le tabelle possono essere **monodimensionali** (vettori, una sola colonna) o **bidimensionali** (matrici, più colonne).
Ogni **cella** di una tabella è come se fosse una **variabile**. **!** Le tabelle bidimensionali non sono ancora state implementate.

Esempi:

Tabella monodimensionale (1D, vettore)

1
32
2
432432
32.1

Tabella bidimensionale (2D, matrice)

1	4324	523
32	21	65
2	76	25
432432	6347	6
32.1	654	3

A differenza di altri linguaggi le tabelle sono “prefabbricate” e pronte all’uso.
Ogni tabella ha 200 righe e 16 colonne (anche se non vengono disegnate). Ogni cella può contenere numeri reali o stringhe di testo.
Ad esempio, la colonna a sinistra contiene un nome di persona (stringa di testo)
e la colonna a destra l’età di quella persona (numero reale):

“Paolo”	43
“Maria”	21
“Giovanni”	10
“Matteo”	56
“Silvia”	8

Non ci sono costrutti per utilizzare le tabelle ma solo delle funzioni specifiche.

Creare una tabella

crea tabella --> (NOME:)

Con questa funzione creiamo una tabella assegnandoli un nome univoco
(non può essere un nome già usato per una variabile o una costante).

Modificare il valore di una cella di una tabella

modifica tabella --> (NOME:) (RIGA:) (COLONNA:) (NUOVO VALORE:)

Con questa funzione si può assegnare o modificare un valore contenuto in una cella, specificando il NOME della tabella e identificando la cella tramite gli argomenti RIGA e COLONNA. Se la tabella è monodimensionale si può omettere l’argomento COLONNA. L’argomento NUOVO VALORE sarà il nuovo valore contenuto in quella cella.

Leggere e memorizzare in una variabile il contenuto di una cella

variabile = ottieni numero da tabella --> (NOME:) (RIGA:) (COLONNA:)
variabile = ottieni testo da tabella --> (NOME:) (RIGA:) (COLONNA:)

Con queste funzioni si può leggere il contenuto di una cella e memorizzarlo in una variabile
Se la tabella è monodimensionale si può omettere l'argomento COLONNA.
Se si tenta di leggere un numero come una stringa di testo, il numero viene trasformato in una stringa (esempio: 23 diventa "23", non rappresenta più un numero ma due caratteri tipografici, "2" e "3"). Viceversa è possibile trasformare una stringa che contiene cifre in un numero reale (gli eventuali caratteri testuali vengono rimossi).

Modificare/dichiarare velocemente un'intera tabella **! Questa funzione non è ancora stata implementata.**

riempi tabella --> (NOME:) (VALORI:)

con questa funzione è possibile riempire tutte le celle di una tabella usando il simbolo "|" per separare i valori di una colonna .
(non sono ammessi spazi tra un valore e l'altro)

Esempio:

riempi tabella --> (NOME: tabella_esempio) (COLONNA 1: 10|34|23|58|"mario"|21|25) (COLONNA 2: 21|"Luisa"|14)

il risultato sarà questa tabella:

10	21
34	"Luisa"
23	14
58	
"Mario"	
21	
25	

Distruggere una tabella

distruggi tabella --> (NOME:)

distruggendo una tabella si libera spazio nella memoria (migliorando le prestazioni).
Ciò consente anche di riassegnare il nome utilizzato ad una nuova tabella.
È buona prassi distruggere una tabella quando non è più necessaria.

Disegnare una tabella:

disegna tabella --> (NOME:) (X:) (Y:) (RIGHE:) (COLONNE:) (COLORE SFONDO:) (COLORE TESTO:) (COLORE LINEE:) (SCALA:)

disegnare una tabella è utile per correggere gli errori nel codice e per avere una visione più chiara dei dati e della loro organizzazione

Esportare e importare una tabella **! Questa funzioni non sono ancora state implementate.**

esporta tabella --> (NOME:)
importa tabella --> (NOME:)

è possibile importare ed esportare tabelle in formato html (compatibili con Excel, browser web e altri programmi).

ABBREVIARE IL CODICE

abbrevia --> (QUESTO:) (CON:)

Con la funzione speciale *abbrevia* è possibile sostituire intere parti di codice con una parola, delle cifre o una sigla preceduta dal **simbolo @**. Questo rende più veloce la scrittura e, se ben utilizzata, in determinate situazioni rende il codice più snello e leggibile, soprattutto se contiene *parti ridondanti* la cui scrittura non può essere evitata tramite *cicli per* e *finche*.

Ad esempio:

```
p1=ottieni stato tasto virtuale --> (NOME: alfa)
p2=ottieni stato tasto virtuale --> (NOME: beta)
p3=ottieni stato tasto virtuale --> (NOME: gamma)
```

Può essere abbreviato in questo modo:

```
abbrevia -->  (QUESTO: "=ottieni stato tasto virtuale --> ") (CON: "tasto")
p1 @tasto (NOME: alfa)
p2 @tasto (NOME: beta)
p3 @tasto (NOME: gamma)
```

o addirittura cosi:

```
abbrevia -->  (QUESTO: "=ottieni stato tasto virtuale --> (NOME:") (CON: "t")
p1 @t alfa)
p2 @t beta)
p3 @t gamma)
```

Questa funzione è un'arma a doppio taglio: se da un lato rende il codice meno ingombrante (ATOMIC lo è per sua natura) dall'altro lato è facile abusarne scrivendo del codice criptico senza rendersene conto; a quel punto ci si ritrova davanti ad un codice di difficile lettura ed interpretazione, in primo luogo per gli altri ma a lungo termine anche per se stessi. Per evitare queste situazioni è buona prassi scrivere la funzione abbrevia **subito prima della parte di codice in cui viene utilizzata**; questo non è un limite tecnico, anche se viene scritta alla fine del codice essa funzionerà comunque perché viene sempre eseguita con la massima priorità prima della tokenizzazione. Anche scriverla nell'evento CICLO CONTINUO non è un problema: verrà comunque eseguita una sola volta.
L'importante è rendere il più possibile intuibile e chiaro lo schema di sostituzione.

Un altro vantaggio oltre alla riciclabilità del codice è la sua **manutenibilità**: in questo modo si può evitare di editare decine di righe di codice uguali, rendendo più veloce la modifica e la correzione degli errori.

Con questa funzione è anche possibile sintetizzare intere parti di codice. Queste parti possono essere **chiuse**, ovvero "stare in piedi da sole" (stand-alone) o **aperte**, ovvero richiedere un completamento per risultare sintatticamente corrette. È importante separare le funzioni con il **simbolo** "**:**" (due punti) **tra una funzione e l'altra** all'interno della stringa e **alla fine della stringa** nel caso sia una abbreviazione aperta.

Esempio:

```
INIZIA
abbrevia -->  (QUESTO: "allora se b=3 allora {disegna linea --> (X 1: 20) (Y 1: 30) (X 2:60) (Y 2:100) (SPESSORE:3) (COLORE: blu)
(TRASPARENZA: 0.75) : disegna cerchio --> (X: 20) (Y: 40) (TRASPARENZA: 0.5)  (SOLO CONTORNO: vero) : ") (CON: "d")

abbrevia --> (QUESTO: "disegna cerchio --> (X: 100) (Y: 100) (COLORE: rosso)  (RAGGIO: 20) : disegna cerchio --> (X: 200) (Y: 200)
(RAGGIO: 30) (COLORE: verde)") (CON: "c")

CICLO CONTINUO
//@d vuol dire disegna alcune cose
se a=1 @d (COLORE: rosso) (X: 300) } }
se a=2 @d (COLORE: verde) } }
se a=3 @d (COLORE: blu) } }

@c
```

Scrivendo solo i due caratteri “@d” possiamo disegnare una linea blu e un cerchio che a seconda dei casi assume un colore diverso (in un caso anche la posizione) ma solo se la variabile b è uguale a 3. Da notare che il colore del cerchio viene specificato dopo e che è necessario inserire anche un’altra parentesi graffa in ogni posizione in cui viene richiamata l’abbreviazione per completare correttamente il codice. Questo è un esempio di codice sintetizzato **aperto**, che richiede un completamento, ovvero che “non sta in piedi da solo”.

Scrivendo invece “@c” possiamo disegnare due cerchi senza bisogno di specificare altro. In questo caso però non possiamo aggiungere altri argomenti all’ultima funzione *disegna cerchio* perché il codice è **chiuso**; ha però il vantaggio di “stare in piedi da solo”.

L’esempio è banale ma può certamente farvi intuire gli innumerevoli utilizzi di questa praticissima funzione. In alcuni casi l’abbreviazione può essere una valida *sostituta all’utilizzo delle classi*, anche se meno flessibile.
La sostituzione del codice grazie all’abbreviazione avviene una sola volta all’avvio del programma, prima ancora dell’evento inizia, a prescindere dalla posizione in cui è scritta; perciò è impossibile intervenire sulle abbreviazioni durante l’esecuzione del programma (nemmeno con la GOD MODE) perché dopo aver svolto il loro lavoro terminano di esistere sostituendo se stesse con il codice che rappresentano.

Un uso alternativo: l’abbreviazione per definire costanti

L’abbreviazione può anche essere utilizzata per definire **costanti** personalizzate. Questo metodo migliora le prestazioni rispetto all’utilizzo delle variabili ma soprattutto è più corretto dal punto di vista logico. Per questa procedura si può utilizzare la forma alternativa: *definisci costante --> (VALORE:) (NOME:)*

Ad esempio:

```
INIZIA
definisci costante --> (VALORE: “53.4”)  (NOME: costante_x)
i=1
CICLO CONTINUO
ripeti per 1000 volte
{
disegna cerchio --> (X: @costante_x) (Y: i)
aumenta i di 1
}
```

È più performante e corretto di:

```
INIZIA
variabile_x=53.4
i=1
CICLO CONTINUO
ripeti per 1000 volte
{
disegna cerchio --> (X: variabilie_x) (Y: i)
aumenta i di 1
}
```

Questo perché @costante_x viene sostituita direttamente nel codice sorgente con il valore 53.4 mentre la variabile_x, essendo appunto una variabile, può variare, ovvero essere modificata in qualsiasi momento: il computer è obbligato a ricalcolarla ogni trentesimo di secondo (anche se non viene mai modificata nel corso della sua esistenza!), questo calcolo si fa relativamente pesante soprattutto se viene eseguito ciclicamente in un ciclo *ripeti per* o *finche*.

La forma *definisci costante* è equivalente ad *abbrevia*; esiste perché è semanticamente più adatta al contesto.

Ad esempio, il risultato di questo codice è uguale al primo riportato in questa sezione:

```
INIZIA
abbrevia --> (QUESTO: “53.4”) (CON: costante_x)
i=1
CICLO CONTINUO
ripeti per 1000 volte
{
disegna cerchio --> (X: costante_x) (Y: i)
aumenta i di 1
}
```

CONSIDERAZIONI FINALI E IPOTESI FUTURE DI SVILUPPO

ATOMIC è un linguaggio **interpretato** quindi è potenzialmente multipiattaforma, se il progetto prende piede sicuramente verranno rilasciate versioni per Mac e Linux; da valutare più attentamente la realizzazione di un'interprete per Android e iOS. Più difficile, ma non impossibile, è la realizzazione di un'interprete per browser web (HTML5). Se il porting su Mac e Linux è indubbiamente giustificabile e vantaggioso, altrettanto non si può dire dei sistemi operativi mobile: il fine ultimo di Atomic è quello di essere un linguaggio didattico, se si esclude a priori la possibilità di scrivere codice su mobile (fattibile ma estremamente scomodo e sconveniente) l'unico motivo per creare interpreti mobile è stimolare l'utente ad utilizzare il linguaggio per poter vedere le proprie creazioni anche al di fuori di un ambiente desktop (cosa che in se non aggiunge una grande valenza educativa).
La criticità per quanto riguarda le piattaforme web e mobile sono le performance. L'attuale interprete di Atomic è stato realizzato in linguaggio GML in tempi brevissimi grazie a Game Maker Studio, del quale condivide alcune logiche e alcune funzioni. Questo farà storcere il naso agli sviluppatori più navigati ma l'utilizzo di Game Maker Studio ha il vantaggio di rendere molto facile ampliare il progetto: scrivere funzioni per Atomic in GML attualmente è molto semplice, quindi anche i programmatori meno esperti possono collaborare al progetto scrivendo e proponendo le loro funzioni. L'unico svantaggio di questo approccio di sviluppo molto veloce e user friendly è che il codice scritto non è ottimizzato: i token finali che vengono letti dal programma sono simili al codice originale, l'ideale invece è trasformare i token in bytecode per incrementare le prestazioni. Tuttavia come già scritto l'obiettivo di Atomic non è quello di essere competitivo dal punto di vista delle prestazioni ma è quello di diventare il miglior linguaggio didattico possibile.

Un linguaggio in italiano al giorno d'oggi non è in controtendenza? In questi anni si assiste ad un'internazionalizzazione sempre più spinta e l'inglese viene insegnato sempre "di più e prima" (ormai a partire dalla scuola dell'infanzia!) tuttavia la propria lingua madre è indubbiamente la migliore per imparare concetti nuovi. La logica della programmazione per alcuni può risultare difficile, apprenderla direttamente in inglese di certo non migliora le cose e inoltre presuppone una buona conoscenza della lingua, cosa che non è scontata, soprattutto in età evolutiva. In questo modo si separano due competenze specifiche: la comprensione di una lingua, e la comprensione di una sintassi, una maniera di scrivere propedeutica alla programmazione. Non è comunque escluso che il linguaggio implementi anche una traduzione in inglese: in questo modo diventerebbe un linguaggio utile a consolidare la lingua straniera e renderebbe ancor meno traumatico il passaggio ai veri linguaggi di programmazione. Potrebbe addirittura essere tradotto in altre lingue per essere diffuso al di fuori dell'Italia.

Attualmente il codice sorgente di Atomic non è disponibile, verrà reso interamente **open source** solo quando sarà abbastanza maturo, solo se otterrà interesse e solo una volta creata una community di sviluppatori, docenti e ricercatori, interessati ad ampliare il progetto.
Nel frattempo è pubblico il codice GML per creare le proprie funzioni, qui sotto sono riportate le due varianti (funzioni "normali" e funzioni "ottieni").

CODICE GML PER SCRIVERE LE PROPRIE FUNZIONI PER ATOMIC

Sono evidenziate in giallo le parti da scrivere/editare.

FUNZIONI "NORMALI"

```
if tok[i]="nome_funzione"
{
//conta gli argomenti scritti dall'utente
conta_argomenti();

//scrivi il valore di default degli argomenti (massimo 16 argomenti)
arg1="…";
arg2="…";
//   …
//fino al massimo di 16 argomenti
arg16="…";

//controlla gli argomenti in base alle etichette (x = numero totale argomenti)
controlla_x_argomenti("NOME ARGOMENTO 1","NOME ARGOMENTO 2", … )

//usa gli argomenti per definire cosa fa la funzione
…
}
```

ESEMPIO:

```
//DISEGNA CERCHIO
if tok[i]="disegna_cerchio"
{
conta_argomenti();
//scrivi il valore di default degli argomenti
arg1="150";
arg2="150";
arg3="100";
arg4="0";
arg5=c_black;
controlla_6_argomenti("X","Y","RAGGIO","SOLO CONTORNO","COLORE","TRASPARENZA")

//usa gli argomenti per definire cosa fa la funzione
draw_set_alpha(arg6)
draw_set_color(arg5);
draw_circle(real(calculate_string(arg1)),real(calculate_string(arg2)),real(calculate_string(arg3)), real(calculate_string(arg4)));
draw_set_color(c_black);
draw_set_alpha(1);
}
```

La funzione **calculate_string** è fondamentale per scrivere le proprie funzioni: permette di calcolare le espressioni regolari presenti nei token (e quindi negli argomenti). Per esempio se *arg1* contiene l'espressione "10*2+(1+a/5)"con a=15 il risultato di *calculate_string(arg1)* sarà "24". Notare che il risultato è sempre una stringa, quindi deve essere convertito con la funzione *real()* se si vuole utilizzare il risultato come valore numerico.

FUNZIONI "OTTIENI"

```
if tok[i+1]="f=" && tok[i+2]="ottieni_nome_funzione"
{
//conta gli argomenti scritti dall'utente
conta_argomenti();

//scrivi il valore di default degli argomenti (massimo 16 argomenti)
arg1="…";
arg2="…";
//   …
//fino al massimo di 16 argomenti
arg16="…";

//controlla gli argomenti in base alle etichette (x = numero totale argomenti)
controlla_x_argomenti("NOME ARGOMENTO 1","NOME ARGOMENTO 2", … )

//usa gli argomenti per definire cosa fa la funzione
…

//definisci il risultato
output_result=string( …  );
aggiorna_variabile_tramite_funzione();
}
```

ESEMPIO

```
//OTTIENI UN VALORE COMPRESO TRA QUESTI
if tok[i+1]="f=" && tok[i+2]="ottieni_un_valore_compreso_tra_questi"
{
//arg0 = variabile a cui applicare il valore ottenuto
arg0=tok[i];

conta_argomenti();

//scrivi il valore di default degli argomenti
arg1="0";
arg2="0";

controlla_2_argomenti("VALORE_1","VALORE_2")

//usa gli argomenti per definire cosa fa la funzione e applica il risultato
output_result=string(random_range(real(calculate_string(arg1)),real(calculate_string(arg2))));
aggiorna_variabile_tramite_funzione();
}
```