ATOMIC

Parti a razzo a programmare!!

v 1.10 - 01/10/2018

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 d'utilizzo.

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 sull'utilizzo di un pc e di matematica.

Consiglio: prima di leggere questo manuale "gioca" un po' con Atomic. Apri i kit pronti, guardali funzionare e poi prova a modificarli.

Buona lettura!

INDICE

CODING O PROGRAMMAZIONE?

Solitamente al termine **coding** (inteso come l'uso di strumenti per la programmazione didattica) viene associata solo la programmazione visuale praticata dai bambini, mentre con il termine programmazione ci si riferisce alla programmazione professionale praticata tramite linguaggi testuali dagli informatici. Atomic è un ibrido che tenta di rompere questa linea di confine. Molti studenti, soprattutto se sosterranno un indirizzo di studio di tipo scientifico, avranno a che fare con i linguaggi di programmazione testuali, anche se non diventeranno degli informatici. Per tale motivo in questo manuale non si fa distinzione tra il termine coding e programmazione (intesa come programmazione didattica).

PERCHÉ USARE UN LINGUAGGIO TESTUALE?

La programmazione informatica è un mondo enorme e nonostante il **pensiero computazionale** sia universale le sue regole sintattiche possono cambiare sensibilmente da linguaggio a linguaggio. Esistono già vari strumenti che permettono di fare coding didattico come Scratch e code.org ma che utilizzano la programmazione visuale. Utilizzando linguaggi visuali è impossibile commettere errori sintattici; questo è un grosso vantaggio per la produttività ma, dato che "sbagliando s'impara", anche un grosso limite. Il salto tra la programmazione visuale e quella testuale (ovvero quella dei "veri" linguaggi di programmazione) consiste proprio nell'apprendere la sintassi di quel linguaggio: in quel momento si passa dall'avere una serie di opzioni da poter incastrare tra loro in modo intuitivo al panico di ritrovarsi di fronte ad un foglio bianco (letteralmente!).

Atomic affianca un minimo di programmazione visuale alla piena libertà sintattica della programmazione testuale; per questo vuole posizionarsi come "gradino mancante" nella scala dell'apprendimento della programmazione.

ATOMIC È UN LINGUGGIO PIÙ AVANZATO RISPETTO ALLA PROGRAMMAZIONE VISUALE?

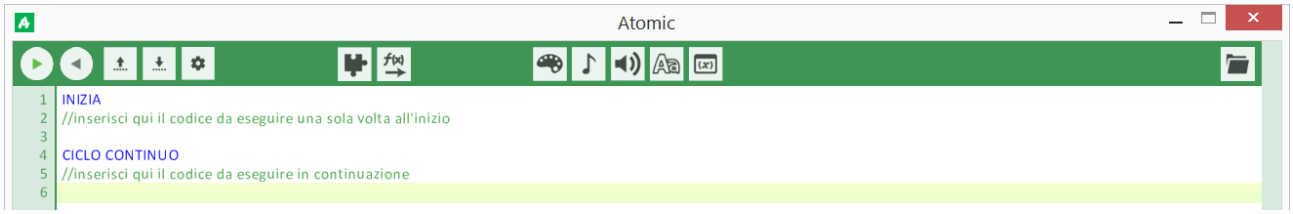
No, è semplicemente diverso. Ogni linguaggio, che sia visuale o testuale, offre caratteristiche e vantaggi diversi. È possibile iniziare ad utilizzare Atomic dopo aver imparato un linguaggio visuale come è possibile iniziare a programmare da zero con Atomic.

Non c'è un età minima consigliata per iniziare a programmare con un linguaggio testuale; l'unico limite rispetto ad un linguaggio visuale è la capacità di utilizzare in modo più o meno fluido la tastiera.

AMBIENTE DI SVILUPPO INTEGRATO

Avviando l'interprete di Atomic si hanno a disposizione solo due opzioni: **Mini editor** ed **Esegui file**.

Cliccando su **Esegui file** si sceglie 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. Oltre a delle comode interfacce per inserire frammenti di codice possiede solo le funzionalità base di un semplice editor di testo (carica, salva, annulla con CTRL+Z, copia/incolla con CTRL+C/V). **Il suo utilizzo è altamente sconsigliato quando si lavora con un codice lungo oltre le 30-40 righe.**

Le prime 5 icone a sinistra sono:

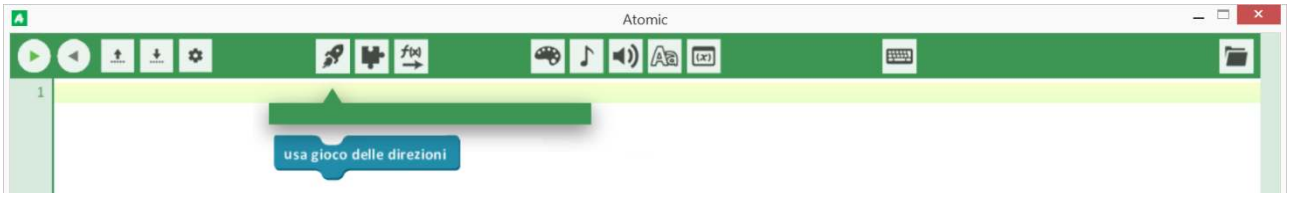
- **Esegui**: esegui il codice che hai scritto
- **Esci**: torna all'interfaccia principale
- **Apri**: apri un file di testo
- **Salva**: salva il lavoro in un file di testo
- **Opzioni**: apri le opzioni

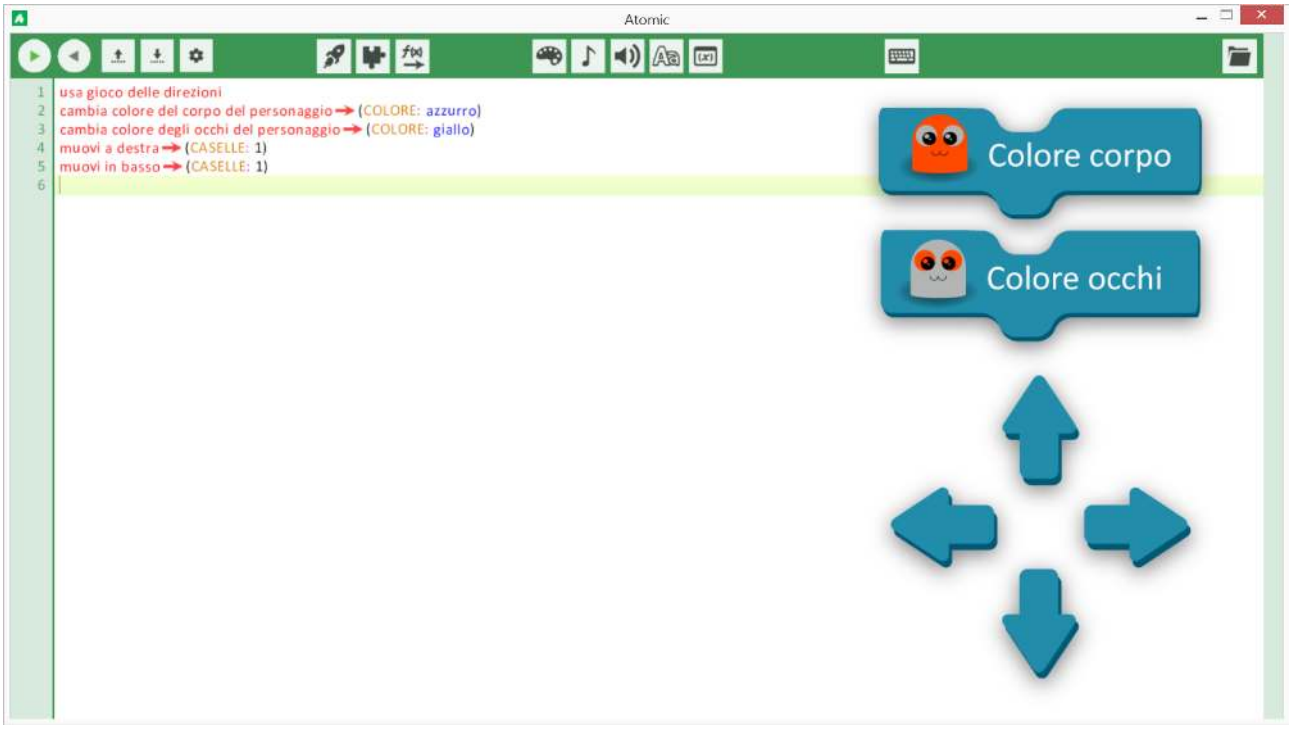
Le icone successive permettono di inserire frammenti di codice pronti all'uso e, nel caso di pezzi più lunghi, sono accompagnati da commenti che spiegano il loro funzionamento e spiegano come devono essere completati.

Le interfacce si rifanno alla **programmazione a blocchi**, simili a quelle utilizzate in **Scratch** e **Code.org**.

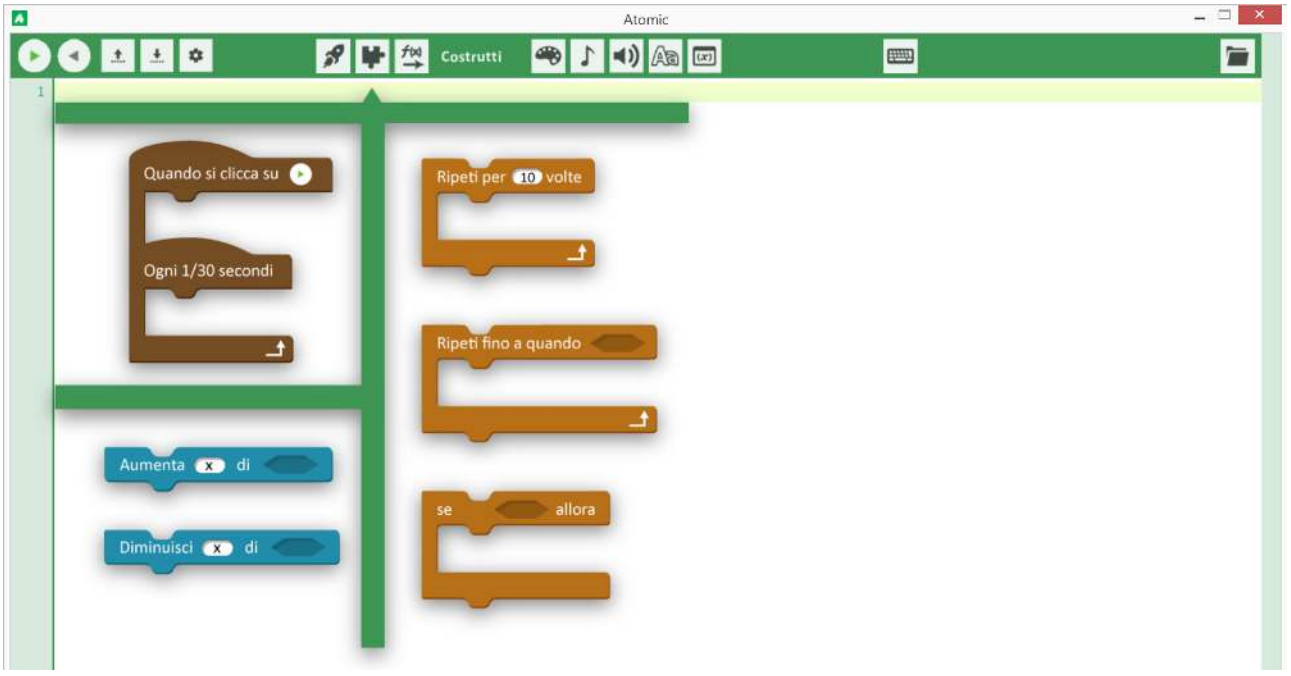
Tuttavia, a differenza di **quest'ultimi Atomic non utilizza la programmazione a blocchi ma converte istantaneamente il blocco selezionato in codice e lo inserisce nella posizione corrente del puntatore.**

L'icona **Kit pronti** permette di inserire pezzi di codice pronti per essere eseguiti e modificati. Alcuni kit comprendono funzioni speciali per un determinato gioco o esercizio e possono far comparire delle interfacce aggiuntive che semplificano la scrittura o la modifica del codice (es. gioco delle direzioni)

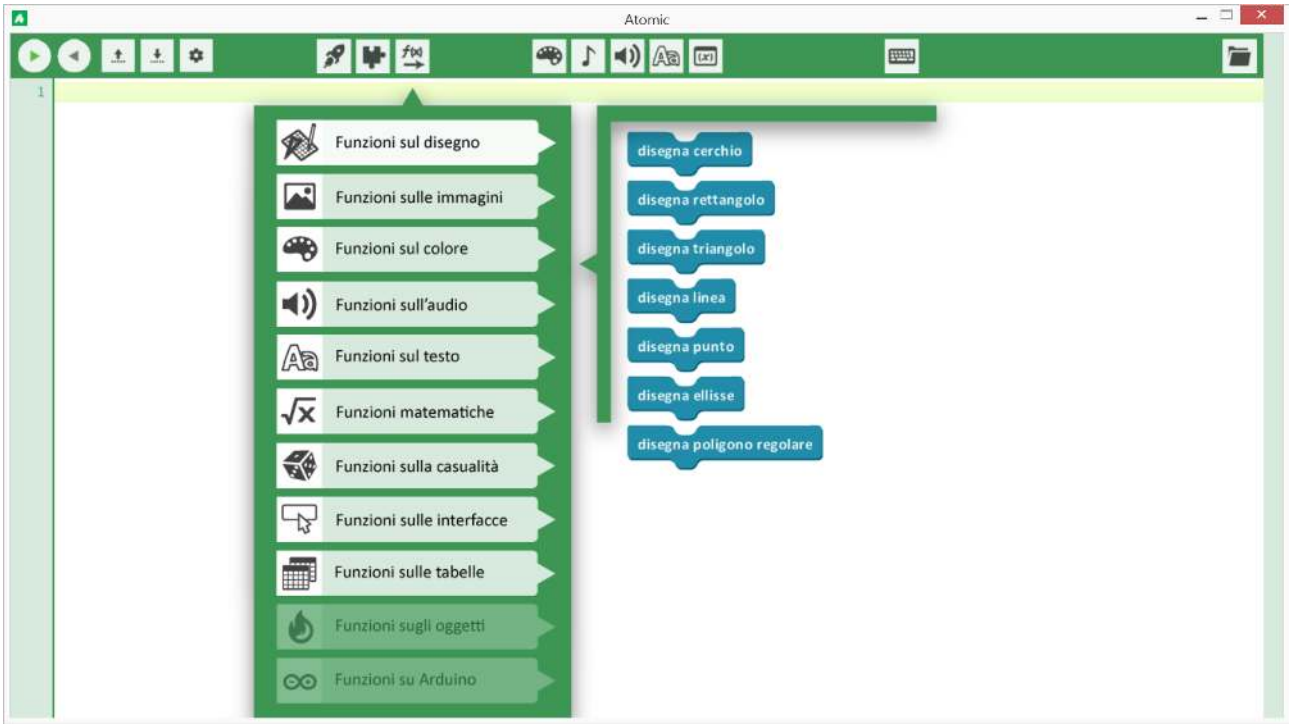




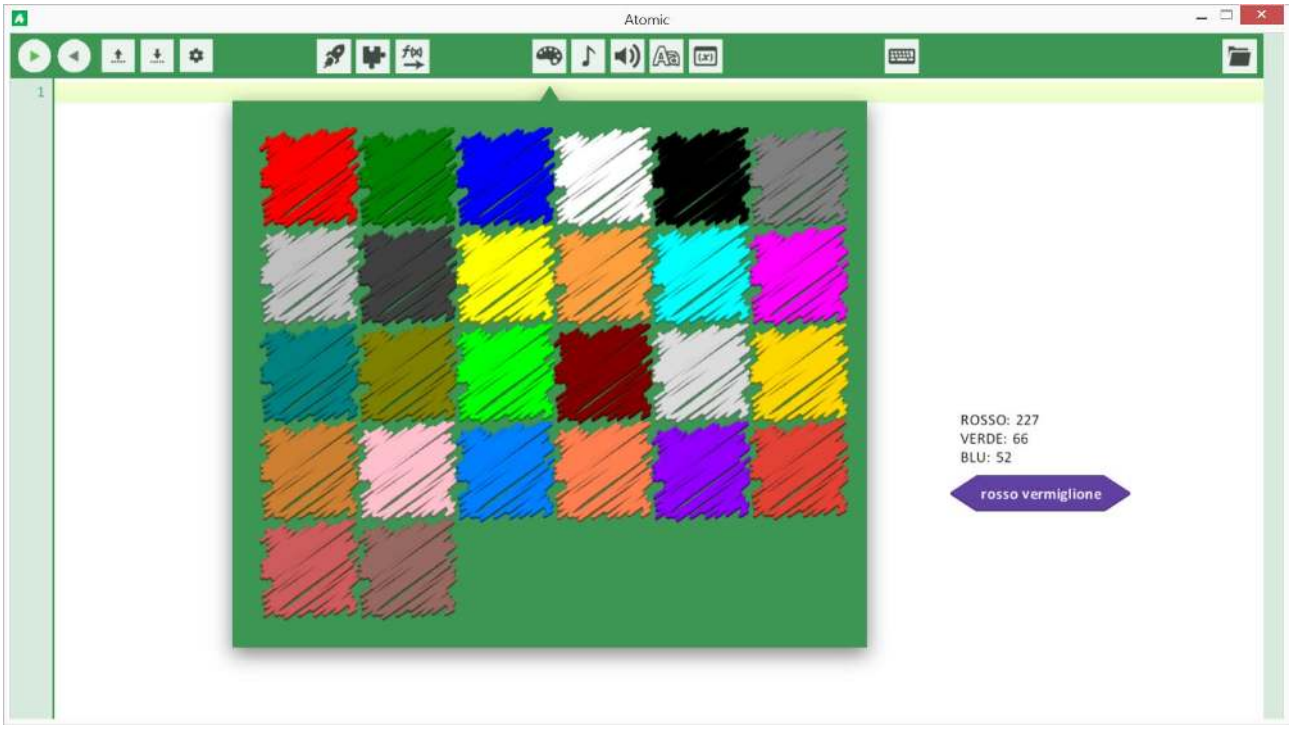
L'icona **Costrutti** permette di inserire le strutture sintattiche di base di Atomic, gli "scheletri" su cui basare le proprie creazioni.



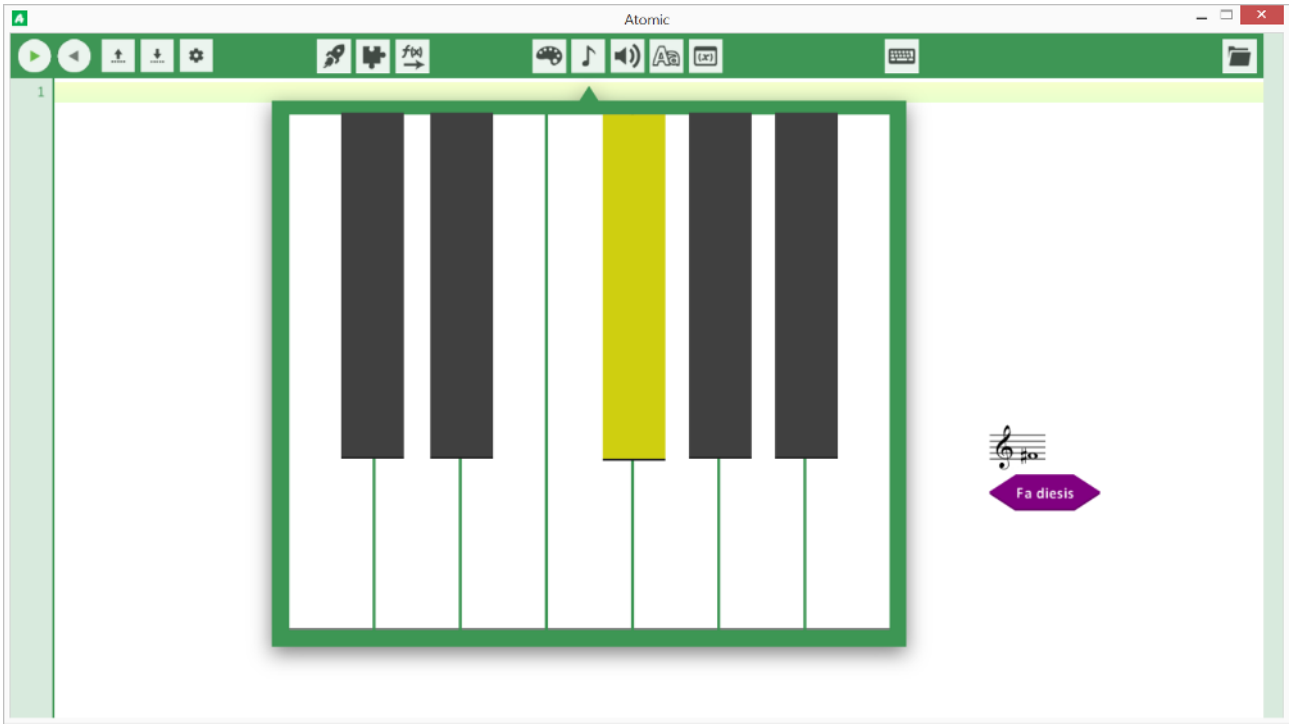
L'icona **Funzioni** permette di inserire la maggior parte delle funzioni di Atomic. Le funzioni sono divise per categoria, in modo da avere una buona panoramica delle "azioni" disponibili.



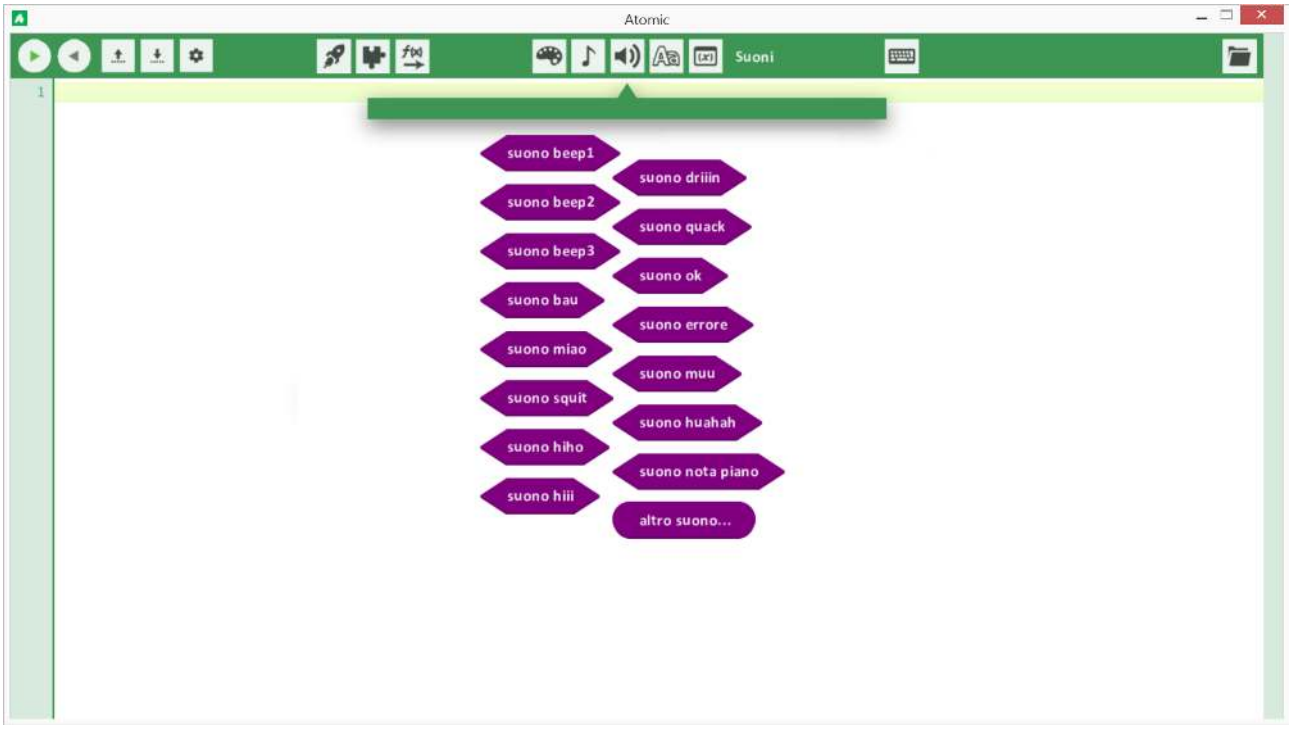
L'icona **Colori** permette di visualizzare ed inserire i colori di base disponibili ed esaminarne la composizione.



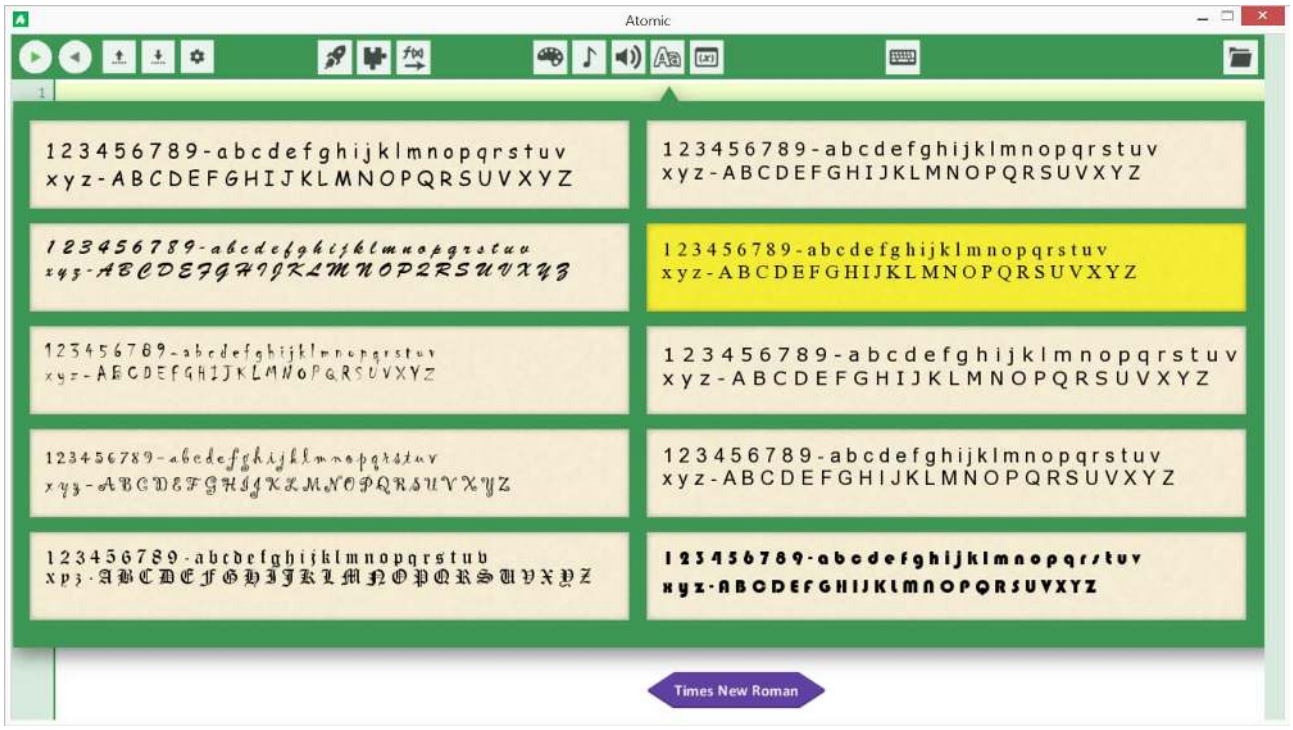
L'icona **Note musicali** permette di inserire e ascoltare l'anteprima delle note musicali.
Premendo il tasto **Ctrl** è possibile suonare la tastiera senza inserire le note.



L'icona **Suoni** permette di ascoltare e inserire i suoni integrati in Atomic.

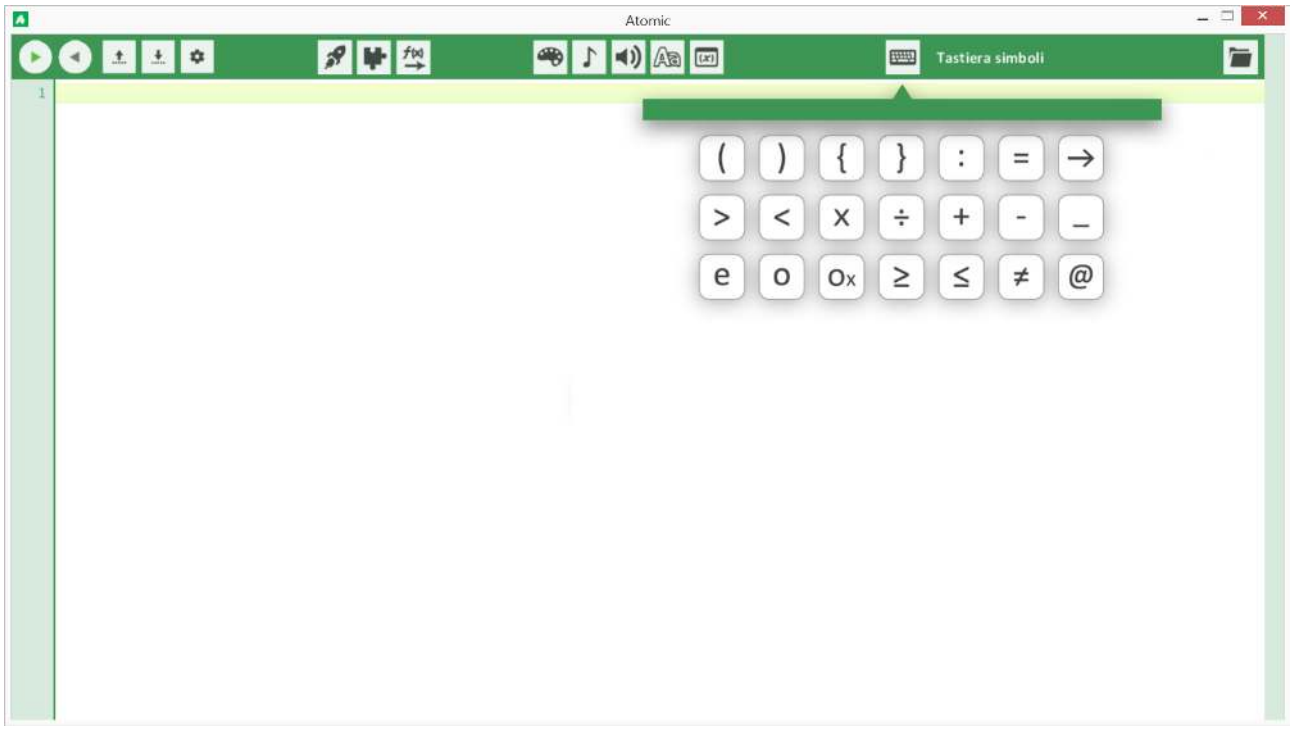


L'icona **Caratteri tipografici** permette di visualizzare e inserire caratteri tipografici (font).

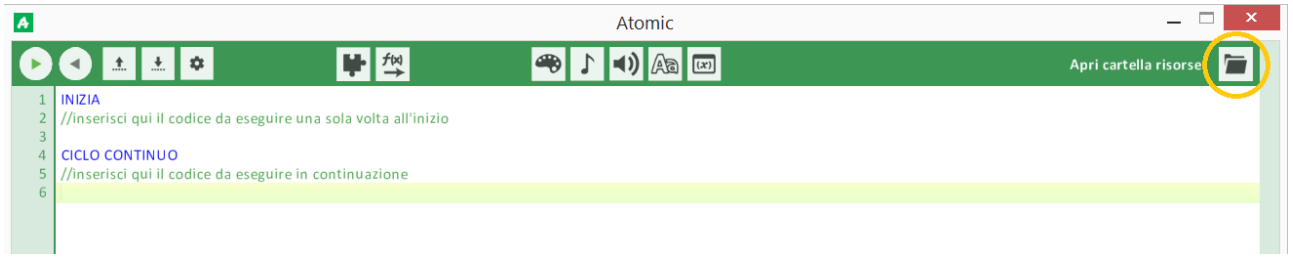


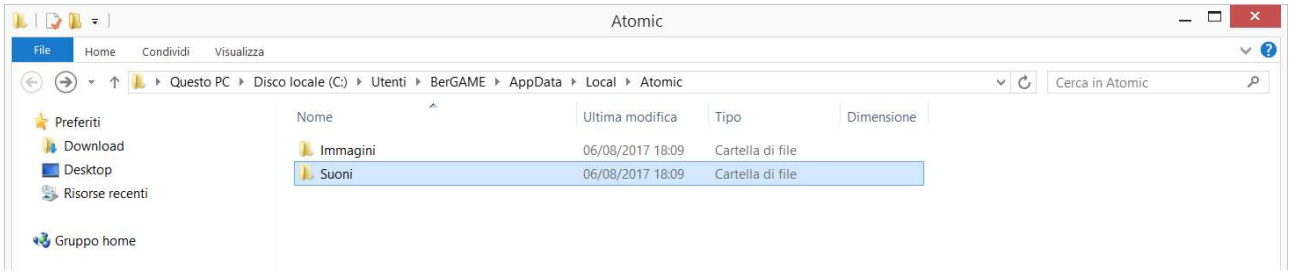
L'icona **Variabili** permette di visualizzare e inserire le variabili integrate.

L'icona **Tastiera simboli** permette di inserire agevolmente i simboli di Atomic nella posizione del cursore.



L'icona **Apri cartella risorse** apre la cartella dove è possibile inserire immagini, animazioni, suoni e musica personalizzati.





UN AMBIENTE SICURO

Malware

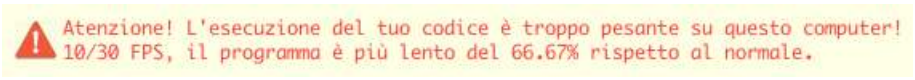
Atomic è un ambiente di sperimentazione sicuro. Con Atomic non è possibile creare programmi che danneggino il computer (sia involontariamente che volontariamente), poiché l'unica cartella al quale Atomic ha accesso è la sua cartella delle risorse. Se dei file sospetti compaiono in questa cartella vengono immediatamente cancellati.

Evita il blocco del computer

Nel caso il computer si blocchi poiché sta tentando di eseguire un ciclo infinito interrompe automaticamente l'esecuzione del codice dopo 25000 iterazioni. Nel caso di un caricamento eccessivo di risorse (immagini e audio) Atomic blocca il caricamento.

Avvisa in caso di prestazioni troppo basse

Se l'esecuzione del codice è pesante (causando un rallentamento) viene notificato durante l'esecuzione del programma. Se il calo di prestazioni è significativo e perdura nel tempo il programma viene arrestato per evitare il surriscaldamento della macchina.



Filtro Internet

Atomic può accedere ad internet ma un filtro interno impedisce di visualizzare e scaricare materiale da siti pornografici, siti che contengono malware o altri siti non adatti ai minori. La lista nera e l'algoritmo di filtraggio sono costantemente aggiornati, tuttavia non garantiamo che sia impossibile eludere il blocco.

In caso di tentato accesso a materiale pornografico viene mostrato un avviso deterrente.

Segnalazioni

Per qualsiasi problema riscontrato relativo alla sicurezza potete segnalare il caso scrivendo a info@bergame.eu .

TIPI DI DATI

Un tipo di dato identifica l'insieme di valori che una variabile può assumere e le operazioni che si possono effettuare su quell'insieme di valori.

In Atomic esistono solo due tipi di dato: **numero** e **testo**.

Un numero (numero reale) è un qualsiasi dato di tipo numerico, sia intero che decimale.

Esempi:

1, 2, 3.56, 257, 1000, 24525.23

I numeri possono essere manipolati tramite espressioni e da un gran numero di funzioni.

La maggior parte delle funzioni richiedono l'utilizzo di numeri per essere utilizzate.

Un testo (stringa di testo) è un qualsiasi dato testuale racchiuso tra i simboli " " (virgolette).

Esempi:

"Ciao", "Siamo in Italia", "Oggi ci sono 20°C"

I testi possono essere manipolati e disegnati tramite apposite funzioni.

Nelle variabili è anche possibile memorizzare riferimenti a delle risorse più complesse come immagini, suoni, font, percorsi, date… queste risorse vengono memorizzate con riferimenti numerici e quindi, oltre ad essere propriamente utilizzati nelle funzioni a loro dedicate, possono essere trattati come dei normali numeri.

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.

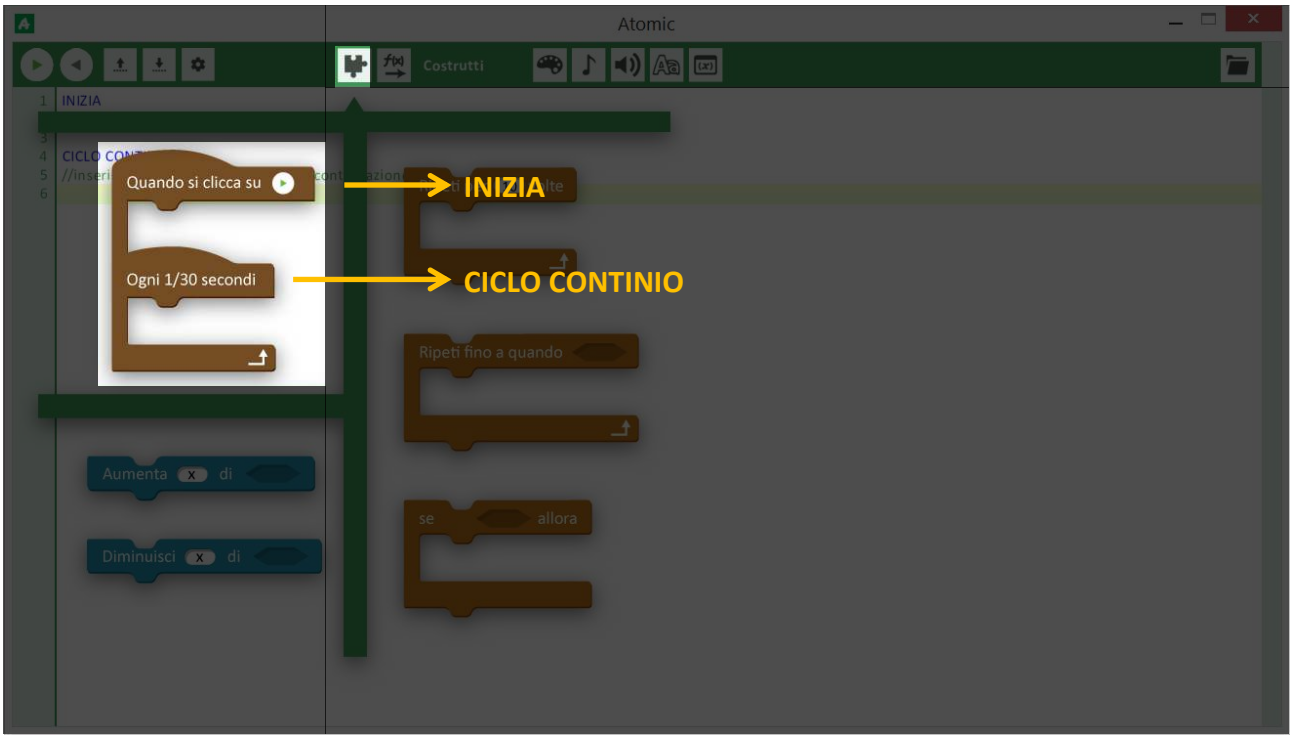
La sintassi da utilizzare è la seguente:

```
INIZIA
… codice …

CICLO CONTINUO
… codice ...
```

```
1  INIZIA
2  //inserisci qui il codice da eseguire una sola volta all'inizio
3
4  CICLO CONTINUO
5  //inserisci qui il codice da eseguire in continuazione
6
```

È possibile inserire gli eventi anche tramite l'icona costrutti:



ESPRESSIONI

Un'espressione è un costrutto che utilizza numeri, variabili e 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 numero, es:

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

Tramite testo, es:

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

Tramite costante, es:

```
1  gatto = vero
2  gatto = pi greco
3  mio_colore = verde
```

Tramite altra variabile, es:

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

Tramite espressione, es:

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

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

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

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

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

```
1  INIZIA
2  gatto = 50
3  tempo_trascorso = 0
4
5  CICLO CONTINUO
6  aumenta tempo_trascorso di 1
7  se tempo_trascorso < 100 allora aumenta gatto di 0.5    disegna cerchio ➔ (RAGGIO: gatto).
8  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

//Queste scritture bizzarre sono valide ma sconsigliate:
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!!!
```

Variabili create automaticamente

In alcuni casi una variabile può essere creata automaticamente da Atomic, vedi le sezioni dedicate all’interfaccia utente (tasti virtuali, interruttori, caselle di spunta, gruppo di opzioni, caselle di testo, barre di controllo).

CHE NOME DARE ALLE VARIABILI?

In matematica spesso le variabili sono nominate con una sola lettera. Questa astrazione dei nomi (il più delle volte convenzionale) può risultare molto comoda a un matematico, un fisico o un chimico… ma di certo non a un bambino o a un ragazzo che si appresta ad imparare i rudimenti della programmazione. A parte rari casi convenzionali (come x e y per le coordinate cartesiane) è sempre meglio nominare una variabile con una o più parole che facciano **riferimento al suo scopo**.

Esempio:

```
problema="Marco possiede 90 figurine.
Il giorno dopo va dal giornalaio e compra 10 pacchetti, ogni pacchetto ne contiene 7.
Nei 10 pacchetti che ha comprato ha trovato 16 doppioni (che si tiene per sè).
Per completare l'album ci vogliono 255 figurine.
Quante figurine possiede ora Marco?
Quante figurine gli mancano per completare l'album?"

//Dati
figurine_iniziali = 90
pacchetti_comprati =  10
figurine_per_pacchetto = 7
doppioni_trovati = 16
figurine_completamento = 255

//Soluzione
figurine_totali =  figurine_iniziali + (pacchetti_comprati * figurine_per_pacchetto)
figurine_utili = figurine_totali - doppioni_trovati
figurine_mancanti = figurine_completamento - figurine_utili

soluzione = "Soluzione: Marco ora posside  <figurine_totali> figurine e gli mancano <figurine_mancanti> figurine per completare l'album"

imposta griglia → (VISIBILE: falso)
disegna testo → (X: 25) (Y: 25) (TESTO: problema) (COLORE: blu)
disegna testo → (X: 25) (Y: 200) (TESTO: soluzione ) (COLORE: verde scuro)
```

I dati e la soluzione del problema sono facilmente leggibili all'interno del codice.

Se sostituiamo i nomi espliciti delle variabili con delle lettere, otteniamo questo:

```
problema="Marco possiede 90 figurine.
Il giorno dopo va dal giornalaio e compra 10 pacchetti, ogni pacchetto ne contiene 7.
Nei 10 pacchetti che ha comprato ha trovato 16 doppioni (che si tiene per sè).
Per completare l'album ci vogliono 255 figurine.
Quante figurine possiede ora Marco?
Quante figurine gli mancano per completare l'album?"

//Dati
a = 90
b = 10
c = 7
d = 16
f = 255

//Soluzione
g =  a + (b * c)
h = g - d
i = f - h

soluzione = "Soluzione: Marco ora posside  <g> figurine e gli mancano <i> figurine per completare l'album"

imposta griglia → (VISIBILE: falso)
disegna testo → (X: 25) (Y: 25) (TESTO: problema) (COLORE: blu)
disegna testo → (X: 25) (Y: 200) (TESTO: soluzione ) (COLORE: verde scuro)
```

Il codice è più breve e più elegante ma la sua comprensione non è immediata.

Per tale motivo in questo manuale raramente vengono usate singole lettere come nomi di variabili. Piuttosto, se il contesto è astratto, vengo usati nomi di animali o comunque delle parole.

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 giù (vero o falso)	no
timer 1, timer 2, timer 3, timer 4, timer 5, timer 6, timer 7, timer 8	Le variabili timer sono 8 variabili che decrescono automaticamente di 1 ogni secondo e possono essere usate per semplificare la gestione del tempo.	si

* "nome" può essere una delle seguenti parole: invio, barra spaziatrice, indietro, ctrl, alt, freccia su, freccia giù, 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, caratteristiche) 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:

```
1 disegna cerchio → (X: 100) (Y: 300) (RAGGIO: 200)
```

che può anche essere scritta senza problemi in questo modo:

```
1 disegna cerchio → (Y: 300) (RAGGIO: 200) (X: 100)
```

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

```
1 disegna cerchio → (Y: 300) (RAGGIO: 200) (X: 100) (COLORE: blu)
```

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

```
1 disegna cerchio → (COLORE: blu)
```

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

```
1 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.

Gli argomenti di una funzione sulla stessa linea possono anche essere racchiusi nella stessa parentesi e separati da una virgola e uno spazio. Inoltre il simbolo "-->" si può omettere. Esempio:

```
1 disegna cerchio (Y: 300, RAGGIO: 200, X: 100, COLORE: blu)
```

Questa sintassi è valida e si avvicina di più ai linguaggi di programmazione più avanzati.

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:)

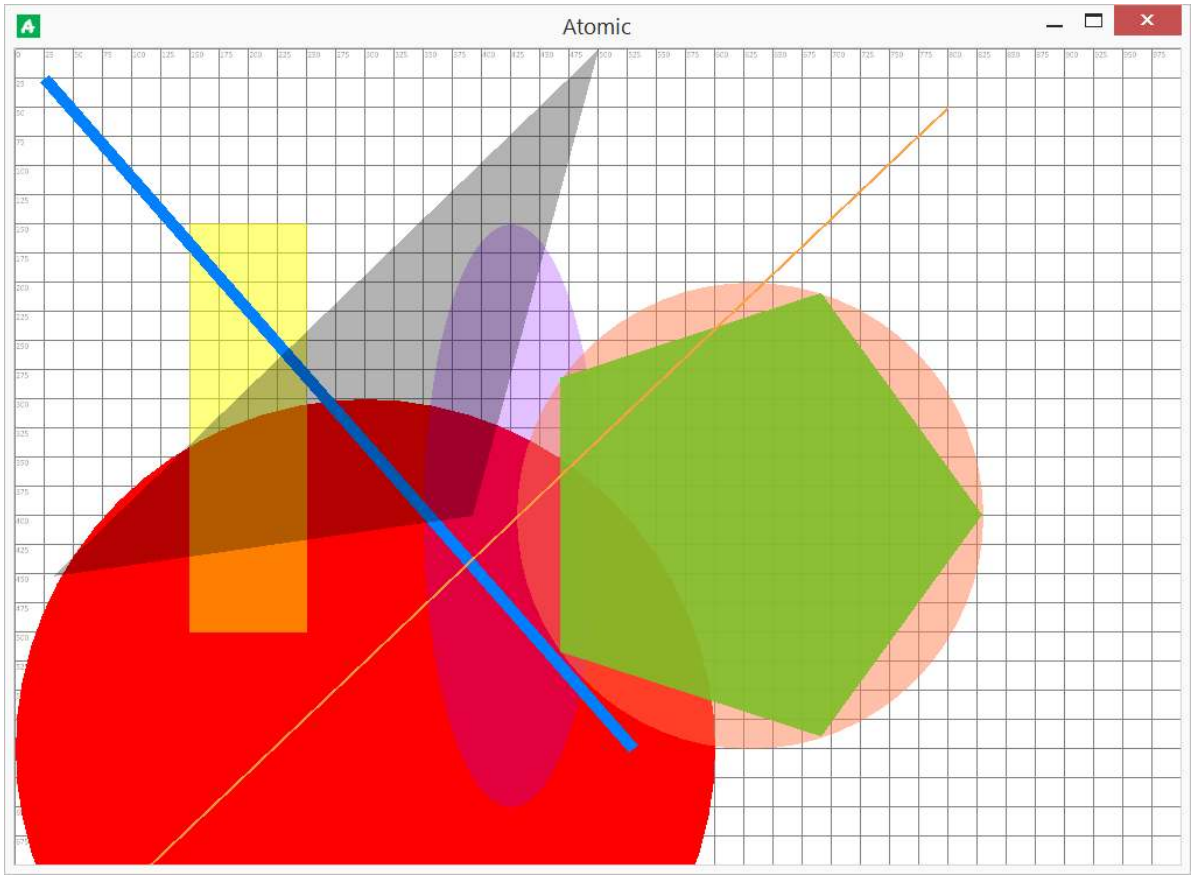
disegna punto --> (X:) (Y:) (COLORE:) (TRASPARENZA:)

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:

```
1 disegna cerchio ➔ (X: 300) (Y: 600) (RAGGIO: 300) (COLORE: rosso)
2 disegna rettangolo ➔ (X 1: 200) (Y 1: 20) (BASE: 100) (ALTEZZA: 350) (RAGGIO: 300) (COLORE: giallo) (TRASPARENZA: 0.5)
3 disegna ellisse ➔ (X 1: 350) (Y 2: 253) (X 2: 500) (Y 2: 650) (COLORE: viola) (TRASPARENZA: 0.25)
4 disegna linea ➔ (X 1: 25) (Y 1: 25) (X 2: 530) (Y 2: 600) (COLORE: azzurro) (SPESSORE: 10)
5 disegna poligono regolare ➔ (NUMERO LATI: 5) (RAGGIO: 300) (X: 630) (Y: 400) (COLORE: verde) (TRASPARENZA: 0.9)
6 disegna cerchio ➔ (RAGGIO: 200) (X: 630) (Y: 400) (COLORE: corallo) (TRASPARENZA: 0.5)
7 disegna linea ➔ (X 1: 800) (Y 1: 50) (X 2: 10) (Y 2: 800) (COLORE: arancione) (SPESSORE: 2)
8 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 sul disegno che richiedono un chiarimento:

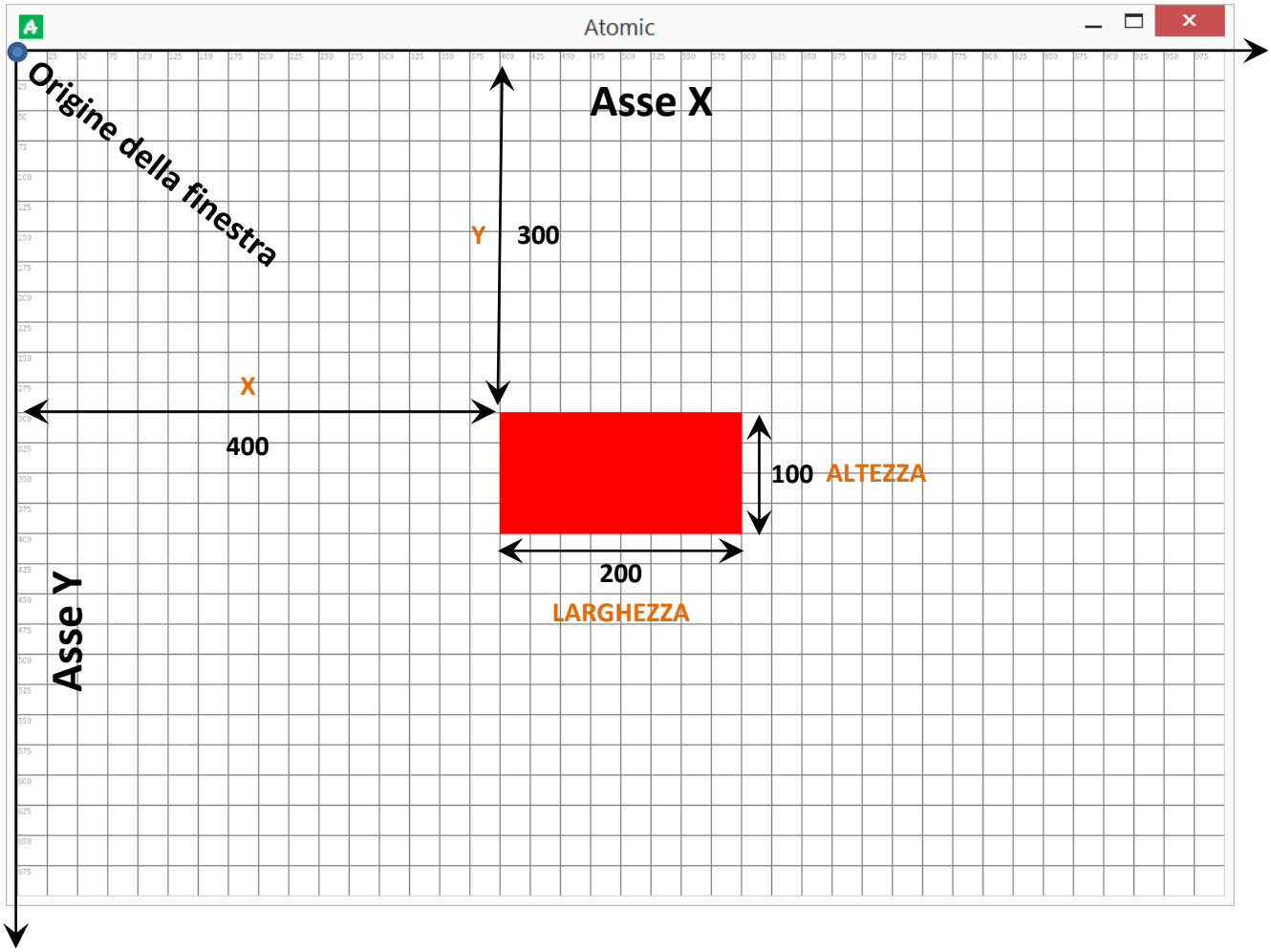
1) **l'origine della finestra** (il piano cartesiano) come in altri ambiti informatici è **in alto a sinistra** e non in basso a sinistra come da convenzione matematica. Questo ribaltamento dell'asse y è comodo soprattutto quando si lavora con il testo.

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.

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) Se si utilizza gli eventi, 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:) (SCALA:) (COLORE:) (TRASPARENZA:) (ROTAZIONE:) (LARGHEZZA CASELLA:) (INTERLINEA:)
(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:

```
1 disegna testo → (TESTO: "Ciao mondo")
```

Esempio più complesso:

```
1 disegna testo → (TESTO: "Ciao mondo") (X: 300) (Y: 100) (COLORE: viola) (TRASPARENZA: 0.65)
2                 (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)

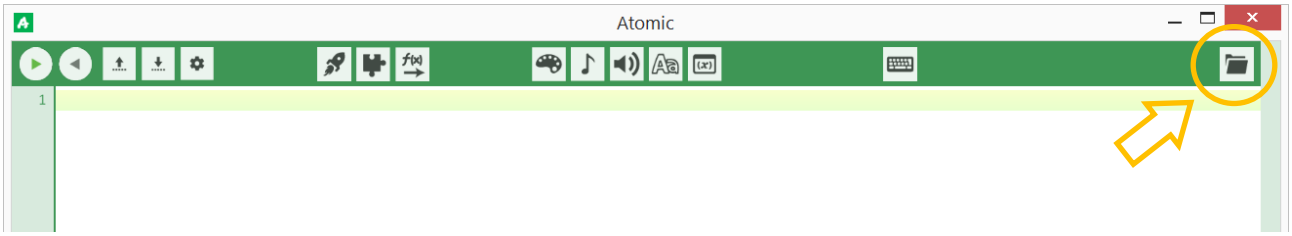
Esempio:

```
1  INIZIA
2  stile_fumetto = ottieni stile testo ➔ (CARATTERE: Comic Sans ) (DIMENSIONE: 80)
3
4  CICLO CONTINUO
5  disegna testo ➔ (TESTO: "Ciao mondo") (STILE: stile_fumetto ) (COLORE: blu )
```

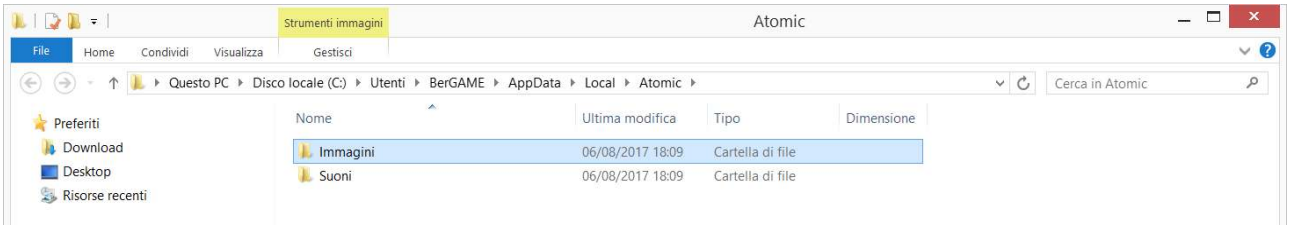
stile_fumetto nell'esempio è la variabile al cui interno viene memorizzato lo stile (è possibile usare qualsiasi nome valido come nome per la variabile). Il simbolo = che precede la funzione indica l'assegnazione di un dato alla variabile, in questo caso un dato complesso **ottenuto** da una funzione.

Funzioni di disegno delle immagini

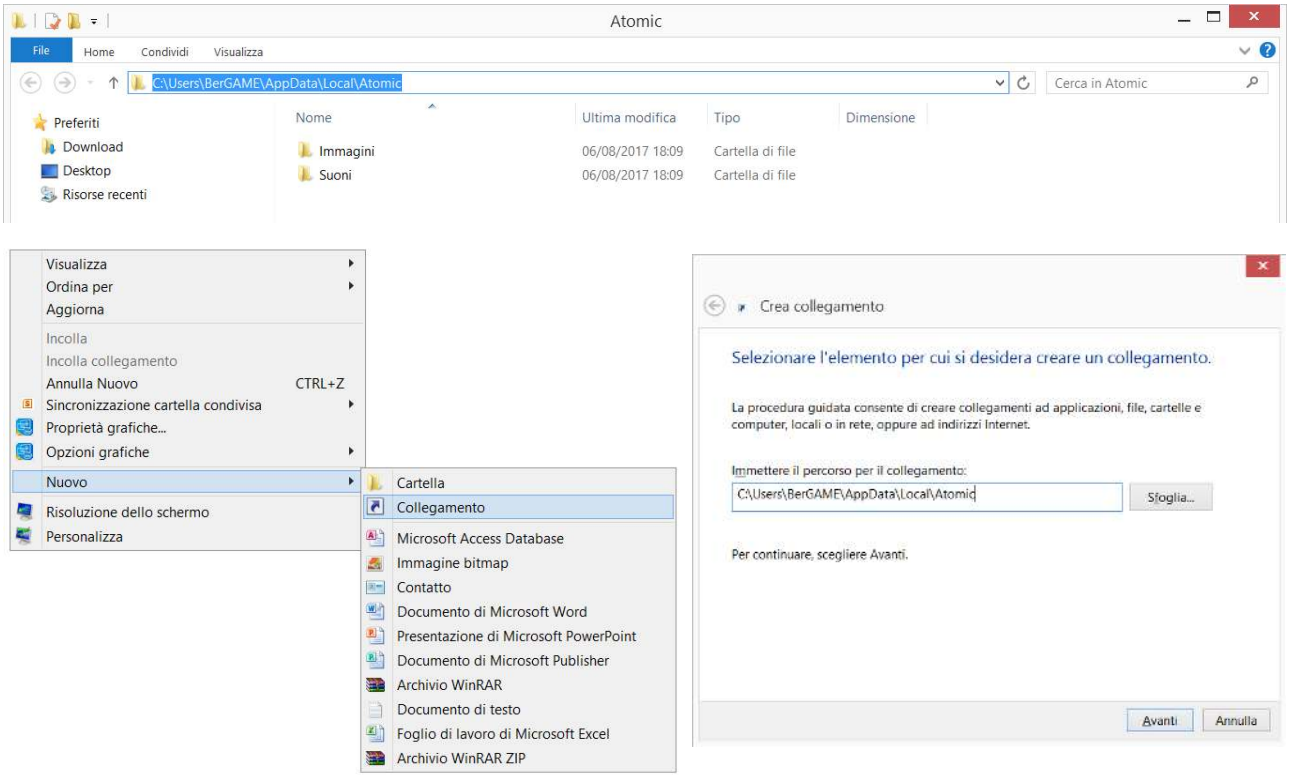
Con la funzione ***disegna immagine*** è possibile disegnare qualsiasi immagine di formato .png e .jpg.
Queste immagini per poter essere disegnate vanno inserite all'interno della **cartella risorse**. È possibile accedere a questa cartella dal mini editor cliccando sull'icona **Apri cartella risorse**.



In alternativa è possibile accedere alla cartella digitando **%LOCALAPPDATA%/Atomic/** sulla barra di Explorer.



Una volta entrati nella 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 immagini di Atomic:

```
1 foto = ottieni immagine ➔ (NOME: "Immagini/fotografia_bella.jpg")
```

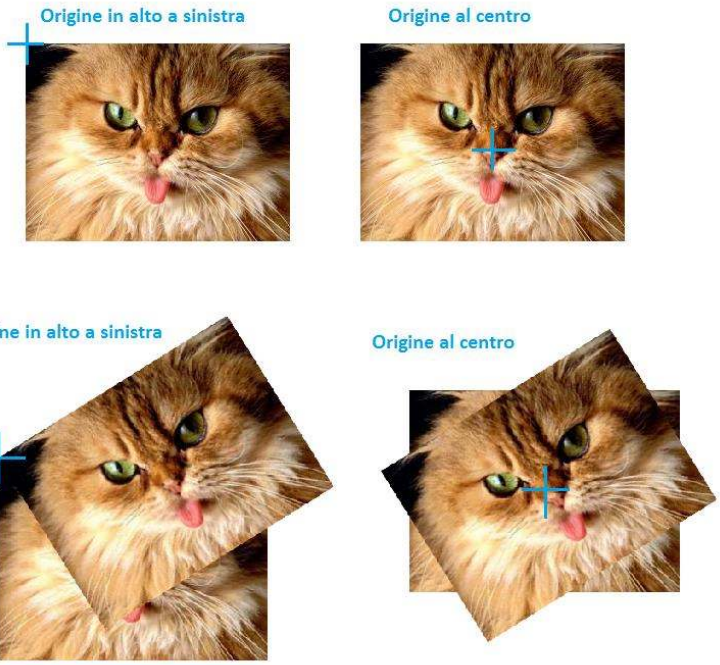
foto nell'esempio è la variabile al cui interno viene memorizzata l'immagine (è possibile usare qualsiasi nome valido come nome per la variabile). Il simbolo = che precede la funzione indica l'assegnazione di un dato alla variabile, in questo caso un dato complesso **ottenuto** da una funzione.

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:

```
1 foto = ottieni immagine ➔ (NOME: "Immagini/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 *Immagini* e all'interno ci inserisco l'immagine *gatto.jpg*, posso richiamare l'immagine *gatto.jpg* in questo modo:

```
1 foto = ottieni immagine ➔ (NOME: "Immagini/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:

```
1 INIZIA
2 foto = ottieni immagine ➔ (NOME: "Immagini/fotografia_bella.jpg")
3
4 CICLO CONTINUO
5 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:

```
1  INIZIA
2  foto = ottieni immagine → (NOME: "Immagini/paesaggio.jpg")
3
4  CICLO CONTINUO
5  disegna immagine → (IMMAGINE: foto)
6  se tasto invio è stato premuto = vero allora 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!".

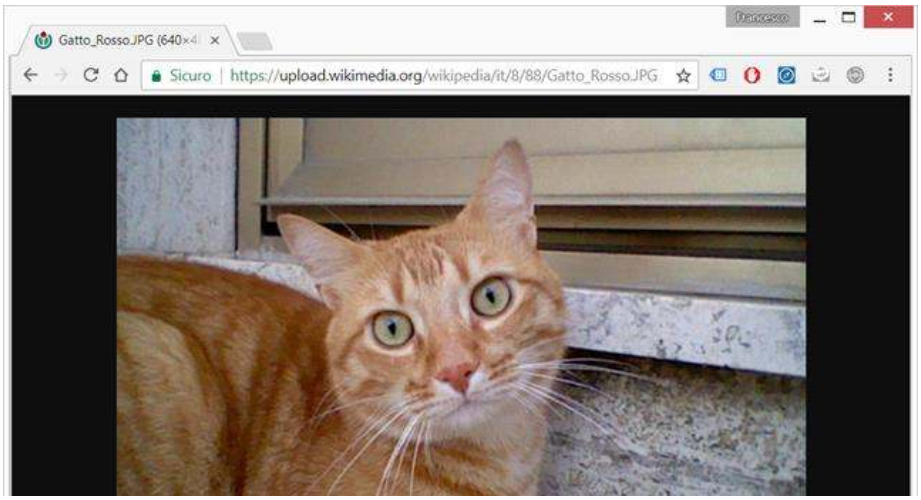
È importante inserire le immagini che ci servono **una sola volta per ogni immagine;** il modo più facile per farlo è inserirle nell'evento INIZIA. Se si vuole inserire un'immagine nell'evento CICLO CONTINUO bisogna stare attenti che l'immagine venga inserita una sola volta altrimenti il programma terminerà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

Caricare immagini da internet

Oltre alle immagini nella cartella risorse è possibile caricare e disegnare immagini presenti sul web (rete internet).
Questa pratica è l'ideale per realizzare progetti che si vogliono condividere, poiché in questo modo basta passare soltanto il codice (file di testo .txt) evitando di passare tutte le immagini utilizzate.
Per utilizzare un'immagine presente sul web bisogna precedere l'indirizzo dalla parola chiave **internet/** ed escludere dall'url il protocollo ("http://" o "https://") e l'eventuale "www.".

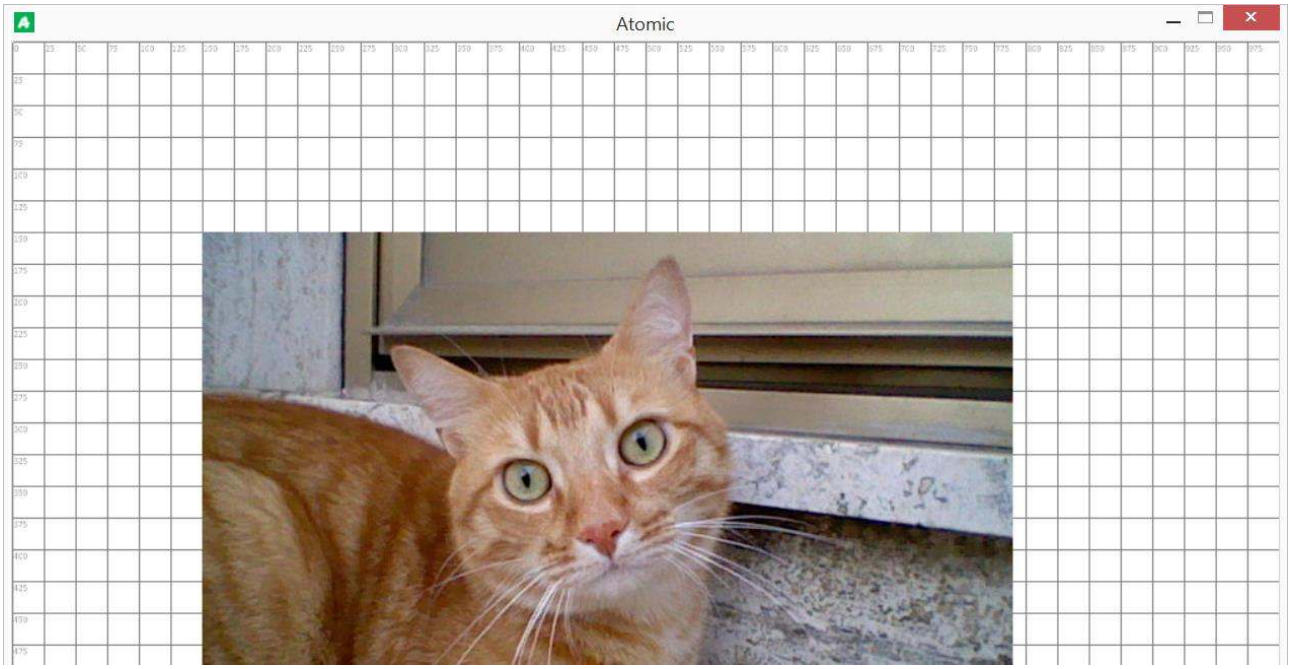
La sintassi sarà quindi: **(NOME: internet/...indirizzo…)**

Se ad esempio si vuole utilizzare l'immagine all'indirizzo https://upload.wikimedia.org/wikipedia/it/8/88/Gatto_Rosso.JPG:



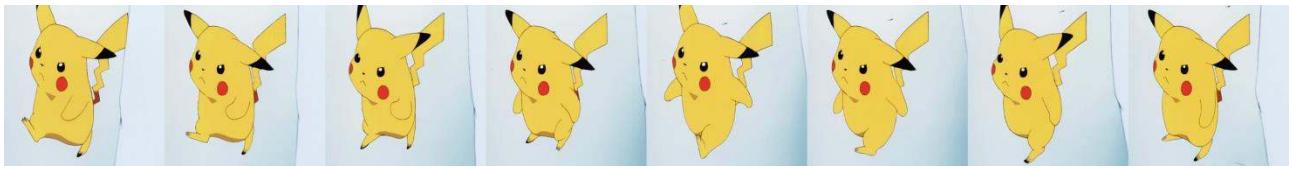
Il codice da utilizzare sarà:

```
1  INIZIA
2  foto = ottieni immagine → (NOME: "internet/upload.wikimedia.org/wikipedia/it/8/88/Gatto_Rosso.JPG")
3
4  CICLO CONTINUO
5  disegna immagine → (IMMAGINE:  foto)
```



Disegnare immagini animate

Per inserire immagini animate dobbiamo utilizzare una striscia (strip) di immagini in sequenza; ad esempio questa, la sequenza di un personaggio che cammina:



Se abbiamo un'animazione in formato ".gif" per utilizzarla dobbiamo prima trasformarla in una striscia in formato ".png". Per farlo è possibile utilizzare vari tool online, ad esempio: https://xattools.firebaseapp.com/ConvertAnimation.html

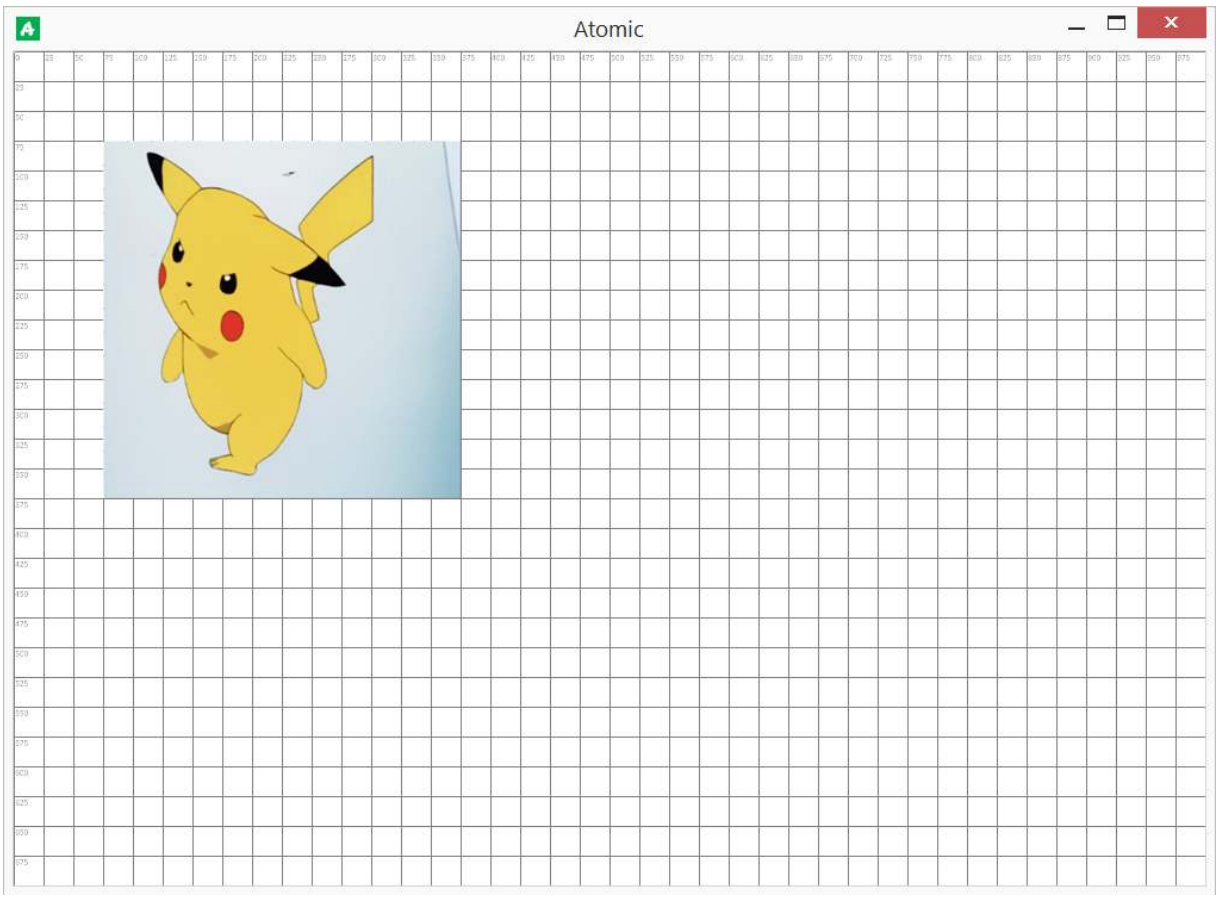
Una volta ottenuta la striscia basta specificare il **numero di fotogrammi** che contiene tramite l'argomento **FOTOGRAMMI** della funzione *ottieni immagine*.

Tramite l'argomento **VELOCITA ANIMAZIONE** della funzione *disegna immagine* è possibile definire la velocità dello scorrimento dei fotogrammi in **fotogrammi al secondo**.

Con questa tecnica potete modificare velocemente animazioni esistenti o creare immagini animate da zero con semplici programmi di grafica (anche con Microsoft Paint!)

Esempio completo:

```
1 INIZIA
2 foto = ottieni immagine ➔ (NOME: "Immagini/pokemon.png") (FOTOGRAMMI: 8)
3 CICLO CONTINUO
4 disegna immagine ➔ (IMMAGINE: foto) (VELOCITA ANIMAZIONE: 15)
```



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:

```
1 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:

```
1 attiva casualità diversa per ogni avvio
2 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.

Oltre ad esercizi specifici in cui vengono utilizzate a scopo dimostrativo, le funzioni matematiche possono tornare utili in tantissime situazioni in cui c'è la necessità di calcolare qualcosa.

A prescindere dalla funzione, Il metodo d'utilizzo è sempre uguale: inserendo dei valori numerici o delle espressioni la funzione restituisce un risultato.

Di seguito le funzioni raggruppate per la difficoltà dell'argomento.

Matematica basilare (Comprensibile durante gli ultimi anni della scuola primaria):

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

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

Matematica facile (Comprensibile durante gli anni di scuola secondaria inferiore)

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

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

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

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

Matematica "avanzata" (Comprensibile durante gli anni di scuola secondaria superiore)

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

ottieni il segno di --> (VALORE:)

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 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 larghezza triangolo dato angolo e lunghezza)

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

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

Funzioni sul testo (stringhe di testo)

Il testo, intesto come tipo di dato, non è altro che una tabella a una sola dimensione (una lista) contenente vari caratteri.

Ad esempio la stringa di testo *"Ciao, questo è un testo"* può essere rappresentata in questo modo:

C	i	a	o	,		q	u	e	s	t	o		è		u	n		t	e	s	t	o	.

Qualsiasi carattere, anche lo spazio vuoto e i simboli di punteggiatura, occupano una cella.

Ogni cella è identificabile da un numero che indica la sua posizione e di conseguenza il carattere che contiene:

Testo	**C**	**i**	**a**	**o**	**,**		**q**	**u**	**e**	**s**	**t**	**o**	**…**
Posizione	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**…**

Di seguito tutte le funzioni per manipolare i testi.

ottieni numero da testo --> (TESTO:)

La funzione *ottieni numero da testo* converte un testo in un numero eliminando gli eventuali caratteri non numerici.
Ad esempio converte "5" in 5, "Ho 10 anni" in 10 e "X:200 Y:350" in 200350.

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

La funzione *ottieni il simbolo di un testo a una determinata posizione* permette di ottenere il carattere (simbolo) contenuto nella cella della posizione specificata.

Esempio:

```
INIZIA
prova = ottieni il simbolo di un testo a una determinata posizione → (TESTO: "Ciao Atomic!") (POSIZIONE: 4)

CICLO CONTINUO
disegna testo → (TESTO: prova)
```

la funzione restituisce il carattere *"o"* poiché si trova nella cella numero 4.

Testo	**C**	**i**	**a**	**o**		**A**	**t**	**o**	**m**	**i**	**c**	**!**
Posizione	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**

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

Tramite la funzione *ottieni il numero di parole specificate in un testo* si ottienine il numero di occorrenze di una parola, ovvero quante volte compare nel testo. Per parola si intende qualsiasi sequenza di caratteri o anche un singolo carattere.

Esempio:

```
INIZIA
frase = "ciao ciao Atomic... ancora ciao!"
prova = ottieni il numero di parole specificate in un testo → (TESTO: frase) (PAROLA: "ciao")
prova2 = ottieni il numero di parole specificate in un testo → (TESTO: frase) (PAROLA: "a")

CICLO CONTINUO
disegna testo → (TESTO: prova) (Y: 100)
disegna testo → (TESTO: prova2) (Y: 200)
```

La variabile *prova* ottiene il valore 3 (vedi riga gialla), la variabile *prova2* ottiene il valore 5 (vedi riga verde)

c	i	a	o		c	i	a	o		A	t	o	m	i	c	.	.	.		a	n	c	o	r	a		c	i	a	o	!
c	i	a	o		c	i	a	o		A	t	o	m	i	c	.	.	.		a	n	c	o	r	a		c	i	a	o	!

ottieni lunghezza testo --> (TESTO:)

La funzione *ottieni lunghezza testo* restituisce la lunghezza del testo specificato.

Esempio:

```
1  INIZIA
2  testo = "testo di prova"
3  lunghezza = ottieni lunghezza testo ➔ (TESTO: testo)
4
5  CICLO CONTINUO
6  disegna testo ➔ (TESTO: lunghezza)
```

T	e	s	t	o		d	i		p	r	o	v	a
1	2	3	4	5	6	7	8	9	10	11	12	13	14

La funzione memorizza il valore 14 nella variabile *lunghezza*.

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

La funzione *ottieni testo con parola sostituita* restituisce il testo specificato nell'argomento **TESTO** sostituendo tutte le occorrenze della sequenza di caratteri (o un singolo carattere) specificata nell'argomento **PAROLA** con la sequenza di caratteri (o un singolo carattere) specificata nell'argomento **NUOVA PAROLA**.

Esempio:

```
1  INIZIA
2  testo = ottieni testo con parola sostituita ➔
3  (TESTO: "I gatti sono animali. I gatti sono mammiferi. I gatti sono belli.")
4  (PAROLA: "gatti") (NUOVA PAROLA: "cani")
5
6  CICLO CONTINUO
7  disegna testo ➔ (TESTO: testo)
```

I		g	a	t	t	i		s	o	n	o		a	n	i	m	a	l	i	.		i		g	a	t	t	i		s	o	n	o	…
I		c	a	n	i		s	o	n	o		a	n	i	m	a	l	i	.		i		c	a	n	i		s	o	n	o	…

L'esempio disegna il testo "I **cani** sono animali. I **cani** sono mammiferi. I **cani** sono belli."

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

La funzione *ottieni testo con parola sostituita solo la prima volta* restituisce il testo specificato nell'argomento **TESTO** sostituendo solo la prima occorrenza della sequenza di caratteri (o un singolo carattere) specificata nell'argomento **PAROLA** con la sequenza di caratteri (o un singolo carattere) specificata nell'argomento **NUOVA PAROLA**.

Esempio:

```
1  INIZIA
2  testo = ottieni testo con parola sostituita solo la prima volta ➔
3  (TESTO: "I gatti sono animali. I gatti sono mammiferi. I gatti sono belli.")
4  (PAROLA: "gatti") (NUOVA PAROLA: "cani")
5
6  CICLO CONTINUO
7  disegna testo ➔ (TESTO: testo)
```

I		g	a	t	t	i		s	o	n	o		a	n	i	m	a	l	i	.		i		g	a	t	t	i		s	o	n	o	…
I		c	a	n	i		s	o	n	o		a	n	i	m	a	l	i	.		i		c	a	n	i		s	o	n	o	…

L'esempio disegna il testo "I **cani** sono animali. I **gatti** sono mammiferi. I **gatti** sono belli."

ottieni testo con inserito altro testo a una determinata posizione --> (TESTO:) (TESTO INSERITO:) (POSIZIONE:)

La funzione *ottieni testo con inserito altro testo a una determinata posizione* permette di inserire un testo all'interno di un altro. L'argomento **POSIZIONE** indica la cella in cui verrà inserito il primo carattere del testo specificato nell'argomento **TESTO INSERITO**. Il testo originale non viene cancellato ma viene traslato per fare spazio ai nuovi caratteri inseriti.

Esempio:

```
INIZIA
testo = ottieni testo con inserito altro testo a una determinata posizione → (TESTO: "Ciao, come stai?")
                                                                              (TESTO INSERITO: " Francesco")
                                                                              (POSIZIONE: 5)

CICLO CONTINUO
disegna testo → (TESTO: testo)
```

C	i	a	o	,		c	o	m	e		s	t	a	i	?									
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	16	20	21	22	23	24	25
C	i	a	o		F	r	a	c	e	s	c	o	,		c	o	m	e		s	t	a	i	?

La funzione restituisce il testo *"Ciao Francesco, come stai?"*.

ottieni un pezzo di questo testo --> (TESTO:) (POSIZIONE:) (LUNGHEZZA:)

La funzione *ottieni un pezzo di questo testo* permette di ricavare un frammento di testo (sottostringa) specificando una cella da cui iniziare a leggere tramite l'argomento **POSIZIONE** e il numero di celle da leggere tramite l'argomento **LUNGHEZZA**.

Esempio:

```
INIZIA
testo = ottieni un pezzo di questo testo → (TESTO: "messaggio di prova") (POSIZIONE: 4) (LUNGHEZZA: 6)

CICLO CONTINUO
disegna testo → (TESTO: testo)
```

M	e	s	s	a	g	g	i	o		d	i		p	r	o	v	a
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

L'esempio disegna il testo *"saggio"*.

ottieni posizione parola --> (TESTO:) (PAROLA:)

La funzione *ottieni posizione parola* restituisce la posizione della **prima occorrenza** della sequenza di caratteri specificata tramite l'argomento **PAROLA**.

Esempio:

```
INIZIA
testo = ottieni posizione parola → (TESTO: "messaggio di prova") (PAROLA: "prova")

CICLO CONTINUO
disegna testo → (TESTO: testo)
```

M	e	s	s	a	g	g	i	o		d	i		p	r	o	v	a
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

L'esempio disegna il valore 14.

ottieni testo maiuscolo --> (TESTO:)

ottieni testo minuscolo --> (TESTO:)

Le funzioni *ottieni testo maiuscolo* e *ottieni testo minuscolo* restituiscono rispettivamente il testo specificato completamente in maiuscolo o in minuscolo.

Esempio:

```
1  INIZIA
2  testo = ottieni testo maiuscolo ➔ (TESTO: "messaggio di prova")
3
4  CICLO CONTINUO
5  disegna testo ➔ (TESTO: testo)
```

L'esempio disegna il testo "MESSAGGIO DI PROVA".

Inserire espressioni in una stringa di testo

Utilizzando i simboli < > per racchiudere il nome di una variabile o una qualsiasi espressione il suo valore verrà rappresentato all'interno del testo.

Esempio:

```
1  età = 10
2  disegna testo ➔ (TESTO: "Ciao! Io ho <età> anni!")
```

Questo codice disegnerà il testo *"Ciao! Io ho 10 anni!"*

NB: Questa funzionalità sostituisce la funzione *ottieni testo combinato*, obsoleta dalla versione 1.06 e in iter di deprecazione.

Testo a capo

Se si vuole ottenere del testo a capo in una stringa è possibile scrivere al suo interno *(a capo)*.

Esempio:

```
1  disegna testo ➔ (TESTO: "GAME OVER! (a capo) Premi il tasto invio per iniziare una nuova partita")
```

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

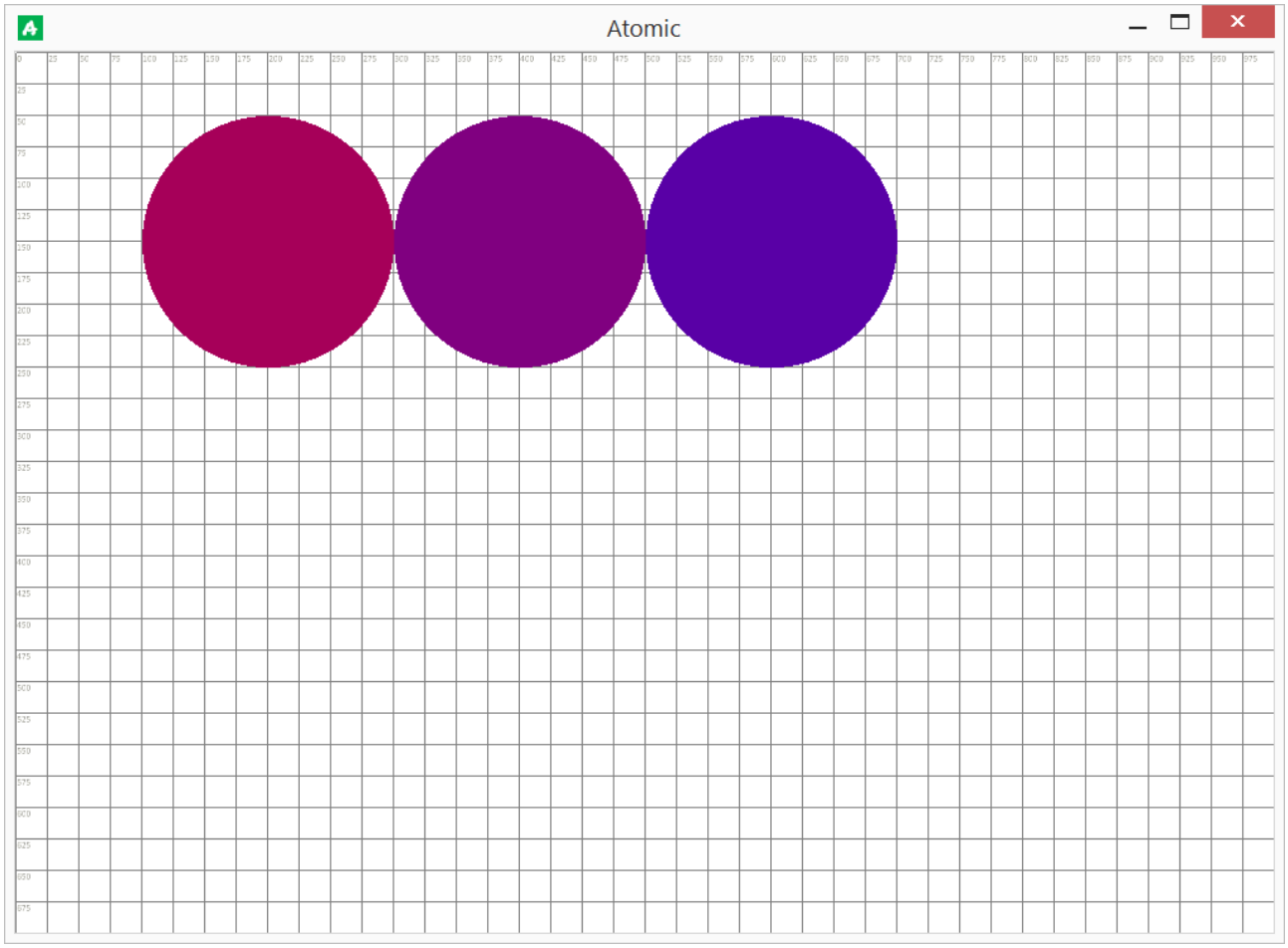
È anche possibile mischiare colori tra loro per ottenerne di nuovi in modo intuitivo.

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.

Esempio:

```
1  //mescolando rosso e blu in proporzioni diverse ottengo varie tonalità di viola/fucsia
2  colore_personalizzato_1 = ottieni miscela colori da ➔ (COLORE 1: rosso) (COLORE 2: blu) (VALORE: 0.35)
3  colore_personalizzato_2 = ottieni miscela colori da ➔ (COLORE 1: rosso) (COLORE 2: blu) (VALORE: 0.5)
4  colore_personalizzato_3 = ottieni miscela colori da ➔ (COLORE 1: rosso) (COLORE 2: blu) (VALORE: 0.65)
5  disegna cerchio ➔ (X: 200) (COLORE: colore_personalizzato_1)
6  disegna cerchio ➔ (X: 400) (COLORE: colore_personalizzato_2)
7  disegna cerchio ➔ (X: 600) (COLORE: colore_personalizzato_3)
```



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. L'argomento ripeti suono permette di ripetere il suono all'infinito (loop) se impostato su vero.

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

è possibile utilizzare qualsiasi numero per l'intonazione ma è consigliabile utilizzare le **note musicali** (vedi sezione costanti).
Come suoni è possibile utilizzare dei suoni già integrati in Atomic (vedi sezione costanti) o utilizzare dei suoni personalizzati.

Esempio completo:

```
1  suona → (SUONO: suono bau) (VOLUME: 0.5) (INTONAZIONE: Fa diesis) (RIPETI SUONO: falso)
```

Per utilizzare un suono o una musica presente sul web bisogna precedere l'indirizzo dalla parola chiave **internet/** ed escludere dall'url il protocollo ("http://" o "https://") e l'eventuale "www.".

Attualmente sono supportati solo i suoni in formato .ogg

Esempio:

```
1  // LINK:    https://upload.wikimedia.org/wikipedia/en/4/45/ACDC_-_Back_In_Black-sample.ogg
2  suona → (SUONO: "internet/upload.wikimedia.org/wikipedia/en/4/45/ACDC_-_Back_In_Black-sample.ogg")
```

Per un maggiore controllo dello streaming è disponibile la funzione **ottieni suono** per caricare il suono e memorizzarlo in una variabile. In questo modo il suono partirà senza ritardi e sarà possibile modificarlo durante l'esecuzione.

Esempio:

```
1  INIZIA
2  musica = ottieni suono → (INDIRIZZO: "internet/upload.wikimedia.org/wikipedia/en/4/45/ACDC_-_Back_In_Black-sample.ogg")
3
4  CICLO CONTINUO
5  se tasto invio è stato premuto allora suona → (SUONO: musica).
```

Altre funzioni sull'audio sono:

interrompi suono --> (SUONO:)

interrompi tutti i suoni

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

ottieni volume principale

ottieni valore volume di --> (SUONO:)

ottieni valore intonazione di --> (SUONO:)

ottieni risultato controllo se sta suonando --> (SUONO:)

modifica suono in esecuzione --> (SUONO:) (INTONAZIONE:) (VOLUME:)

È possibile suonare semplici melodie tramite la funzione *suona melodia*.

suona melodia --> (SUONO:) (NOTE:) (RIPETI MELODIA:) (VOLUME:) (BPM:)

La sintassi da utilizzare per scrivere una melodia è la seguente: (NOTE: nota - nota - nota -)
Ogni trattino divide un **battito (NB: battito è diverso da battuta musicale!)**

Esempio:

```
1 (NOTE: Do - Re - Fa - )
```

Per inserire una pausa in un battito è possibile inserire uno spazio vuoto.

Esempio:

```
1 (NOTE: Do - - Re - )
```

È possibile inserire più pause in modo da rendere più precisa la disposizione temporale delle note.

Esempio:

```
1 (NOTE: Do - - Re - Mi - - - - - Fa diesis - - -La - - Sol )
```

Se una linea è troppo lunga è possibile utilizzare il simbolo $ per andare a capo.

Esempio:

```
1 (NOTE: Do - - Re - Mi - - - - - Fa diesis - - -La - - Sol - - - - - - - - Re - - - Mi - - ->>
2 Fa - - La - - - - - - Si - -Re*2 - - Do*2 )
```

Per modificare la velocità d'esecuzione di una melodia si può modificare l'argomento **BPM** (battiti per minuto).
Tramite l'argomento **SUONO** si indica il suono da utilizzare come strumento musicale.
Tramite l'argomento **RIPETI MELODIA** si indica se la melodia deve essere ripetuta all'infinito (vero) o se deve essere eseguita una sola volta (falso).
Tramite l'argomento **VOLUME** si indica il volume (inteso come gain) con il quale verrà suonata la melodia (1=massimo, 0=muto).

Esempio completo:

```
1 //FRA MARTINO
2 suona melodia➔
3 (SUONO: suono nota piano )
4 (NOTE: Do- -Re- -Mi- -Do- -Do- -Re- -Mi- -Do- -Mi- -Fa- -Sol- - - -Mi- -Fa- -Sol- - - ->>
5 Sol-La-Sol-Fa-Mi- -Do- -Sol-La-Sol-Fa-Mi- -Do- -Do- -Sol- -Do- - - -Do- -Sol- -Do- )
6 (BPM: 100)
7 (RIPETI MELODIA: vero)
8 (VOLUME: 0.75)
```

Questa funzione permette di comporre musica velocemente (per quanto semplice). Inoltre offre una visualizzazione chiara ed immediata della disposizione temporale delle note.

Funzioni sulle interfacce

In Atomic esistono degli oggetti speciali d'**interfaccia grafica per l'utente** (Conosciuta anche come **GUI** – Graphical User Interface).

Questi oggetti semplificano la programmazione dell'interazione utente e sono molto utili per creare applicazioni che manipolano e controllano dati.

Questi oggetti funzionano tutti in modo simile:

- si usa una funzione per creare l'interfaccia necessaria (solitamente nell'evento INIZIA)

- si utilizza il **NOME** dell'interfaccia per ottenere il valore di cui permette il controllo da parte dell'utente.

Le interfacce disponibili sono:

- Tasti virtuali
- Interruttori
- Caselle di spunta
- Barre di controllo
- Gruppi di opzioni
- Caselle di testo

N.B. se volete create un'interfaccia nell'evento **CICLO CONTINUO** fate attenzione che venga creata una sola volta inserendo la funzione all'interno di una condizione.

Esempio:

```
INIZIA
//Obiettivo: creare un singolo tasto durante l'evento CICLO CONTINUO
interfaccia = "non è ancora stata creata"

CICLO CONTINUO
crea tasto virtuale → (NOME: tasto1)
//SBAGLIATO!!! ogni trentesimo di secondo verrà creato un nuovo tasto

se tasto invio è stato premuto { crea tasto virtuale → (NOME: tasto2) }
//SBAGLIATO!!! Ogni volta che premiamo il tasto invio viene creato un nuovo tasto

se tasto invio è stato premuto = vero e interfaccia = "non è ancora stata creata" {interfaccia = "da creare"}
se interfaccia = "da creare" {crea tasto virtuale → (NOME: tasto3) interfaccia="è già stata creata"}
//Ok, questo codice crea il tasto una sola volta quando viene premuto il tasto invio
```

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.

Questo tipo di interfaccia è quella graficamente più versatile, nonché la più comune.

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 utilizzare un tasto virtuale si utilizza il suo NOME come una variabile.

La variabile (che viene creata automaticamente assieme al tasto) contiene 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:

```
1  INIZIA
2  angolo = 45
3  crea tasto virtuale → (NOME: tasto) (ETICHETTA: "Cliccami!") (COLORE SFONDO: rosso) (COLORE TESTO: bianco)
4
5  CICLO CONTINUO
6  se tasto = 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:

```
1  INIZIA
2  crea classe di tasti → (NOME: classe_di_prova) (X: 600) (COLORE SFONDO: nero) (COLORE TESTO: giallo) (ALTEZZA: 50)
3                         (ETICHETTA: "sono un tasto!") (TRASPARENZA: 0.75)
4  crea tasto virtuale → (NOME: tasto_a) (CLASSE: classe_di_prova) (Y: 50)
5  crea tasto virtuale → (NOME: tasto_b) (CLASSE: classe_di_prova) (Y: 150)
6  crea tasto virtuale → (NOME: tasto_c) (CLASSE: classe_di_prova) (Y: 250) (ETICHETTA: "")
7  crea tasto virtuale → (NOME: tasto_d) (CLASSE: classe_di_prova) (Y: 350)
```

Interruttori

Gli interruttori sono degli **oggetti** speciali disegnati all'interno della finestra; possono avere un colore personalizzato e un'etichetta descrittiva.

Il loro funzionamento è simile a quello dei tasti virtuali: possono essere cliccati e ad ogni click l'interruttore passa da attivo a disattivo o viceversa.

Per creare un interruttore si utilizza la funzione *crea interruttore*:

```
crea interruttore --> (NOME: ) (ETICHETTA: ) (X: ) (Y: ) (COLORE: ) (VALORE PREDEFINITO: )
```

L'argomento ETICHETTA è una stringa di testo che viene disegnata a destra dell'interruttore, questa può essere diversa dal nome. L'argomento VALORE PREDEFINITO indica lo stato dell'interruttore al momento della sua creazione (0 o 1, vedi poco più sotto).

Per utilizzare un interruttore si utilizza il suo NOME come una variabile.

La variabile (che viene creata automaticamente assieme all'interruttore) contiene un valore booleano: 1 (vero, ON) o 0 (falso, OFF)

Esempio:

```
1  INIZIA
2  crea interruttore → (NOME: interruttore) (COLORE: azzurro) (VALORE PREDEFINITO: 0)
3                      (ETICHETTA: "interrutore di prova") (X: 300) (Y: 200)
4
5  CICLO CONTINUO
6  se interruttore = 1 allora disegna cerchio .
```

OFF, disattivo, 0

ON, attivo, 1

Questo esempio disegna un cerchio se l'interruttore è attivo.

Caselle di spunta

Le caselle di spunta sono degli **oggetti** speciali disegnati all'interno della finestra; possono avere un un'etichetta descrittiva.

Il loro funzionamento è del tutto identico a quello degli interruttori: possono essere cliccate e ad ogni click la casella passa da attiva a disattiva o viceversa. Le caselle di spunta si differenziano dagli interruttori solo per l'aspetto grafico; di solito vengono utilizzate per settare impostazioni minori rispetto a quelle controllate dagli interruttori.

Per creare una casella di spunta si utilizza la funzione *crea casella di spunta*:

```
crea casella di spunta --> (NOME: ) (ETICHETTA: ) (X: ) (Y: ) (VALORE PREDEFINITO: )
```

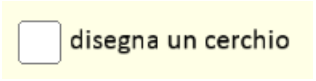
L'argomento ETICHETTA è una stringa di testo che viene disegnata a sinistra della casella, questa può essere diversa dal nome. L'argomento VALORE PREDEFINITO indica lo stato della casella al momento della sua creazione (0 o 1, vedi poco più sotto).

Per utilizzare una casella di spunta si utilizza il suo NOME come una variabile.

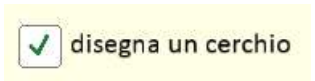
La variabile (che viene creata automaticamente assieme alla casella) contiene un valore booleano: 1 (vero, ON) o 0 (falso, OFF)

Esempio:

```
1  INIZIA
2  crea casella di spunta --> (NOME: impostazione) (VALORE PREDEFINITO: 0)
3                             (ETICHETTA: "disegna un cerchio") (X: 300) (Y: 200)
4
5  CICLO CONTINUO
6  se impostazione = 1 allora disegna cerchio .
```



OFF, disattiva, 0



ON, attiva, 1

Questo esempio disegna un cerchio se la casella ha il segno di spunta.

Barre di controllo

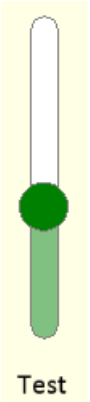
Le barre di controllo sono degli **oggetti** speciali disegnati all'interno della finestra; possono avere un colore personalizzato e un'etichetta descrittiva.

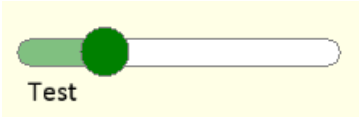
Le barre di controllo permettono di selezionare tramite il mouse un valore compreso tra un massimo e un minimo specificati.

Le barre di controllo possono essere orizzontali o verticali in base alla funzione usata:

crea barra di controllo orizzontale --> (NOME:) (ETICHETTA:) (X:) (Y:) (LARGHEZZA:) (COLORE:)
(VALORE MINIMO:) (VALORE MASSIMO:) (VALORE PREDEFINITO:)

crea barra di controllo verticale --> (NOME:) (ETICHETTA:) (X:) (Y:) (ALTEZZA:) (COLORE:)
(VALORE MINIMO:) (VALORE MASSIMO:) (VALORE PREDEFINITO:)





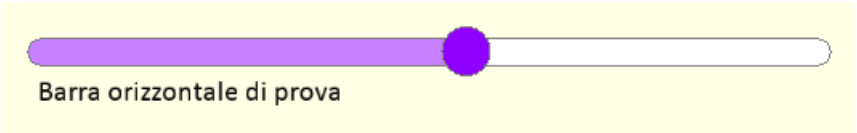
In base alla funzione utilizzata le barre possono avere una LARGHEZZA o un'ALTEZZA personalizzata.
L'argomento ETICHETTA è una stringa di testo che viene disegnata sotto la barra, questa può essere diversa dal nome.
L'argomento VALORE PREDEFINITO indica il valore della barra al momento della sua creazione.

Per utilizzare una barra si utilizza il suo NOME come una variabile.

La variabile (che viene creata automaticamente assieme alla barra) contiene un valore numerico.

Esempio:

```
1  INIZIA
2  crea barra di controllo orizzontale → (X: 50) (Y: 450) (NOME: barra) (VALORE MINIMO: 50) (COLORE:  viola)
3                                        (LARGHEZZA: 500) (VALORE MASSIMO: 500) (VALORE PREDEFINITO: 234)
4                                        (ETICHETTA: "Barra orizzontale di prova")
5
6  CICLO CONTINUO
7  disegna cerchio → (RAGGIO: barra)
```



Questo esempio modifica il raggio di un cerchio in funzione della barra.

Gruppi di opzioni

I gruppi di opzioni sono degli **oggetti** speciali disegnati all'interno della finestra; possono avere un un'etichetta descrittiva.

I gruppi di opzioni sono formati da più opzioni selezionabili. A differenza delle caselle di spunta i gruppi di opzioni permettono di selezionare **una sola casella per gruppo**.

Per creare un gruppo di opzioni si utilizza la funzione *crea gruppo di opzioni*:

```
crea gruppo di opzioni --> (NOME: ) (ETICHETTA: ) (X: ) (Y: )  (VALORE 1:) … (VALORE  8:) (VALORE PREDEFINITO: )
```

L'argomento ETICHETTA è una stringa di testo che viene disegnata a sinistra della casella, questa può essere diversa dal nome.
L'argomento VALORE PREDEFINITO indica l'opzione selezionata al momento della sua creazione (può essere qualsiasi tipo di dato, vedi poco più sotto).

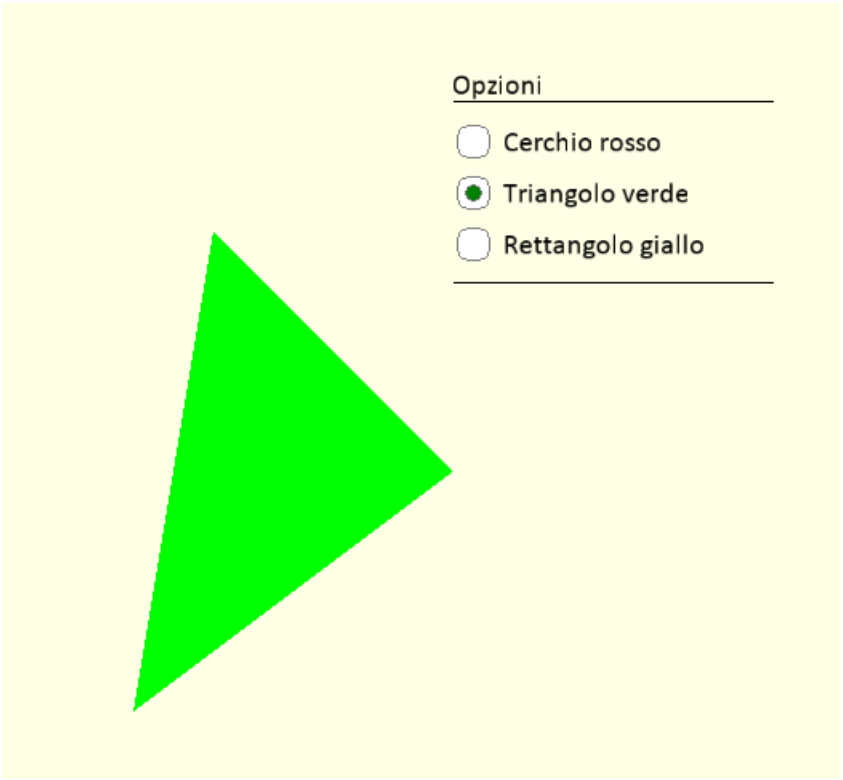
È possibile specificare fino a 8 valori (opzioni) per ogni gruppo; questi valori possono essere numeri o stringhe di testo.

Per utilizzare un gruppo di opzioni si utilizza il suo NOME come una variabile.

La variabile (che viene creata automaticamente assieme al gruppo di opzioni) contiene il valore dell'opzione selezionata.

Esempio:

```
1  INIZIA
2  imposta griglia → (VISIBILE: falso)
3  crea gruppo di opzioni → (ETICHETTA: "Opzioni") (NOME: opzione) (X: 300) (Y: 50)
4                           (VALORE 1: "Cerchio rosso")  (VALORE 2: "Triangolo verde")
5                           (VALORE 3: "Rettangolo giallo") (VALORE PREDEFINITO: "Triangolo verde")
6
7  CICLO CONTINUO
8  se opzione = "Cerchio rosso" { disegna cerchio → (COLORE: rosso) }
9  se opzione = "Triangolo verde" { disegna triangolo → (COLORE: verde) }
10 se opzione = "Rettangolo giallo" { disegna rettangolo → (COLORE: giallo) }
```



Questo esempio disegna una forma geometrica colorata in base all'opzione selezionata.

Caselle di testo

Le caselle di testo sono degli **oggetti** speciali disegnati all'interno della finestra; possono avere un un'etichetta descrittiva e il colore del testo personalizzato.

Le caselle di testo permettono di inserire dati testuali durante l'esecuzione del programma. All'interno di una casella si può anche selezionare, copiare e incollare testo.

Per creare una casella di testo si utilizza la funzione *crea casella di testo*:

```
crea casella di testo--> (NOME: ) (ETICHETTA: ) (X: ) (Y: ) (COLORE: ) (TESTO: ) (LARGHEZZA:)
```

Oppure, se si desidera una casella di testo con più linee, la funzione *crea casella di testo multilinea*:

```
crea casella di testo multilinea --> (NOME: ) (ETICHETTA: ) (X: ) (Y: ) (COLORE: ) (TESTO: ) (LARGHEZZA:) (ALTEZZA:)
```

L'argomento ETICHETTA è una stringa di testo che viene disegnata sopra la casella, questa può essere diversa dal nome.
L'argomento TESTO indica il testo predefinito della casella al momento della sua creazione.
L'argomento COLORE indica il colore del testo.

Per utilizzare una casella di testo si utilizza il suo NOME come una variabile.

La variabile (che viene creata automaticamente assieme alla casella) contiene la stringa di testo contenuta all'interno della casella.

Esempio:

```
INIZIA
imposta griglia → (VISIBILE: 0)
crea casella di testo multilinea → (X: 100) (Y: 100) (NOME: casella)
                                   (ETICHETTA: "Casella di testo")

CICLO CONTINUO
angolo=0
ripeti per 10 volte
{
aumenta angolo di 1
disegna testo → (X: 400) (Y: 200) (TESTO: "TESTO: <casella>") (COLORE: blu)
                (ROTAZIONE: angolo) (TRASPARENZA: angolo/100)
}
```

Questo esempio disegna il testo inserito dall'utente nella casella applicando un semplice effetto grafico di sfocatura da movimento in tempo reale.

È anche possibile impostare il testo all'interno di una casella:

imposta testo in una casella di testo --> (NOME:) (TESTO:)

Esempio:

```
INIZIA
crea casella di testo → (NOME: casella)

CICLO CONTINUO
se tasto invio è stato premuto = vero
{
imposta testo in una casella di testo → (NOME: casella) (TESTO: "")
}
```

Questo esempio cancella il testo inserito dall'utente quando viene premuto il tasto invio.

Eliminare interfacce

È possibile rimuovere qualsiasi interfaccia precedentemente creata (tasti virtuali, interruttori, caselle di spunta, gruppi di opzioni, barre di controllo e caselle di testo) tramite la funzione *distruggi interfaccia***.**

distruggi interfaccia --> (NOME:)

Esempio:

```
INIZIA
crea interruttore → (NOME: esempio)

CICLO CONTINUO
se tasto invio è stato premuto = vero { distruggi interfaccia → (NOME: esempio) }
```

NB: Questa funzione sostituisce la funzione *distruggi tasto virtuale*, obsoleta dalla versione 1.06 e in iter di deprecazione.

Funzioni sulle date e sugli orari

Tramite queste funzioni è possibile ottenere date, informazioni e calcoli tra date e orari.

Questa è la funzione che permette di ottenere la data attuale come valore numerico
(è un valore numerico che può essere utilizzato dal computer per fare calcoli):

ottieni data attuale

Questa è la funzione che permette di ottenere la data attuale come stringa di testo:

ottieni data attuale come testo

Il formato con cui verrà visualizzata sarà: "gg/mm/aaaa – hh:mm:ss"

Esempio:

```
1 data = ottieni data attuale come testo
2 info = ottieni testo combinato ➔ (TESTO 1: "La data di oggi è: ") (TESTO 2: data)
3 disegna testo ➔ (TESTO: info)
```

Queste sono le funzioni che permettono di ricavare un elemento (secondo, minuto, giorno, ecc..) dalla data attuale:

ottieni secondo attuale

ottieni minuto attuale

ottieni ora attuale

ottieni giorno attuale

ottieni nome del giorno attuale

ottieni mese attuale

ottieni nome del mese attuale

ottieni anno attuale

Queste funzioni sono utili per rappresentare una data nel modo che si preferisce, ad esempio: "Giovedì 10 Settembre 2020".

Esempio:

```
1 nome_giorno = ottieni nome del giorno attuale
2 info = ottieni testo combinato ➔ (TESTO 1: "La data di oggi è: ") (TESTO 2: nome_giorno)
3 disegna testo ➔ (TESTO: info)
```

Per ottenere un valore numerico che rappresenta una data è possibile utilizzare la funzione *ottieni una data*:

ottieni una data --> (GIORNO:) (MESE:) (ANNO:) (ORA:) (MINUTO:) (SECONDO:)

Gli argomenti non specificati conterranno il valore 0.

Per controllare se una data è oggi:

ottieni risultato controllo se la data è oggi --> (DATA:)

Restituisce 1 (vero) se la data scritta come argomento è oggi o 0 (falso) se la data non è oggi.

Esempio:

```
1 giorno = ottieni una data ➔ (GIORNO: 6) (MESE: 10) (ANNO: 2020)
2 la_verifica_è_oggi = ottieni risultato controllo se la data è oggi ➔ (DATA: giorno)
3 se la_verifica_è_oggi = vero allora disegna testo ➔ (TESTO: "Oggi è il giorno della verifica di storia!").
4 se la_verifica_è_oggi = falso allora disegna testo ➔ (TESTO: "Oggi non è il giorno della verifica di storia").
```

Per ricavare un elemento (secondo, minuto, giorno, ecc..) da una qualsiasi data è possibile utilizzare la funzione *ottieni elemento da questa data*:

ottieni elemento da questa data --> (ELEMENTO:) (DATA:)

L'argomento DATA richiede una variabile contenente una data. Questa variabile deve essere stata creata precedentemente tramite la funzione *ottieni data attuale* o *ottieni una data*.

L'argomento ELEMENTO definisce cosa vogliamo ottenere dalla data.

Le stringhe supportate sono:

"secondo", "minuto", "ora", "giorno", "nome del giorno", "settimana", "mese", "nome del mese", "anno".

Esempio:

```
1 oggi= ottieni data attuale
2 nome_mese= ottieni elemento da questa data ➔ (DATA: oggi) (ELEMENTO: "nome del mese")
3 mese_attuale= ottieni testo combinato ➔ (TESTO 1: "Siamo nel mese di ") (TESTO 2: nome_mese)
4 disegna testo ➔ (TESTO: mese_attuale)
```

Per confrontare due date è possibile utilizzare la funzione *ottieni risultato confronto date*:

ottieni risultato confronto date --> (DATA 1:) (DATA 2:)

Questa funzione restituisce una di queste stringhe:

"DATA 1 è precedente a DATA 2", "DATA 1 è successiva a DATA 2", "DATA 1 è uguale a DATA 2".

È possibile ottenere la differenza tra due date nell'unità che si preferisce tramite la funzione *ottieni differenza tra due date*.

ottieni differenza tra due date --> (DATA 1:) (DATA 2:) (UNITA:)

Per definire un' unità è possibile utilizzare le seguenti stringhe:

"in secondi", "in minuti", "in ore", "in giorni", "in settimane", "in mesi", "in anni".

Le unità incomplete verranno riportate come frazioni.

Funzioni sui file di testo

Con Atomic è possibile creare, leggere e modificare file di testo (txt, csv, html, css, ini… ecc.) Questi file vengono creati, letti e modificati nella cartella delle risorse di Atomic (%LOCALAPPDATA%/Atomic)

scrivi su questo file --> (NOME:) (TESTO:)

scrivi una nuova linea su questo file --> (NOME:) (TESTO:)

Con queste funzioni è possibile scrivere su un file di testo inserendo del nuovo testo alla fine del documento. Se il file non esiste verrà creato automaticamente.

sovrascrivi su questo file --> (NOME:) (TESTO:)

Con questa funzione è possibile sovrascrivere su un file di testo (il testo precedente viene cancellato). Se il file non esiste verrà creato automaticamente.

ottieni testo da questo file --> (NOME:)

Con questa funzione è possibile ottenere una stringa di testo contenente l'intero testo del file.

ottieni una riga di testo da questo file --> (NOME:) (RIGA:)

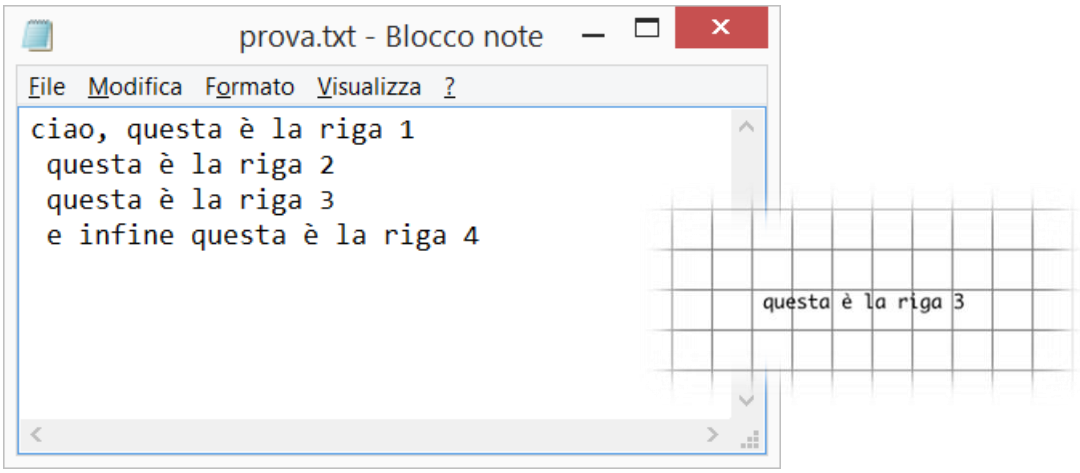
Con questa funzione è possibile ottenere la riga di testo specificata del file.

Esempio:

```
INIZIA
sovrascrivi su questo file → (NOME: "prova.txt")
(TESTO: "ciao, questa è la riga 1
questa è la riga 2
questa è la riga 3
e infine questa è la riga 4")

testo = ottieni una riga di testo da questo file → (NOME: "prova.txt") (RIGA: 3)

CICLO CONTINUO
disegna testo → (TESTO: testo)
```



.

È anche possibile memorizzare il valore delle variabili all'interno del file di testo. Esempio:

```
se game_over = vero
{
scrivi una nuova linea su questo file → (NOME: "classifica.txt") (TESTO: "<nome_giocatore> : <punteggio>")
riavvia il programma
}
```

INTRODUZIONE ALLA PROGRAMMAZIONE AD OGGETTI

Atomic è un linguaggio di programmazione orientato agli oggetti .

Il paradigma ad oggetti offre diversi vantaggi:

- fornisce un sistema "**concreto**" che fa riferimento al mondo reale
- è facile da gestire e **mantenere**
- favorisce la **modularità** e il **riuso del codice** evitando quindi la **ridondanza**

Quando si parla di oggetti è importante distinguere due **elementi** fondamentali:

- **oggetti**
- **esemplari** di oggetti (conosciuti anche come **istanze**, dalla mal traduzione di *instance*)

Un oggetto è astratto, ovvero è la descrizione generica di un tipo di esemplare.

Un esemplare è la concretizzazione di un oggetto.

Un **elemento** può quindi essere un oggetto o un esemplare.

In Atomic il temine "elemento" è usato per riferirsi genericamente a un oggetto o un'esemplare, poiché condividono le stesse caratteristiche, cosi come il temine "animale" può essere usato per riferirsi a un qualsiasi cane o al cane del vicino di casa.

Esempio:

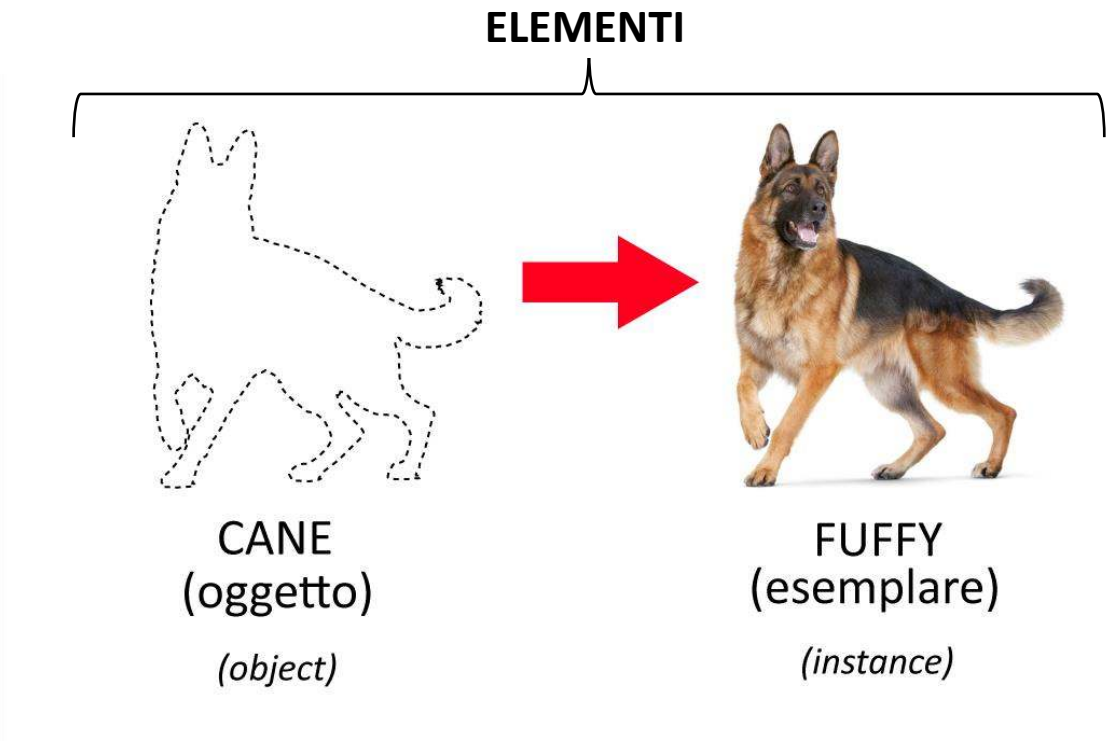
Oggetto: *cane*

Esemplare: *Fuffy*

L'oggetto *cane* è la descrizione generica di un cane.

L'esemplare *Fuffy* è un esemplare di *cane*, unico, identificabile e concreto.

Fuffy e *cane* sono elementi.



EREDITARIETA' E GERARCHIA

Per ogni oggetto si possono definire delle **caratteristiche**, queste caratteristiche sono **ereditarie**.

Gli esemplari ereditano le caratteristiche dell'oggetto a cui appartengono.

Esempio: se la caratteristica COLORE dell'oggetto gatto è arancione tutti gli esemplari di gatto saranno (di base) arancioni.

A loro volta gli oggetti possono appartenere ad altri oggetti sempre più generici ed ereditarne le loro caratteristiche.

In questi casi l'oggetto contenuto viene chiamato **oggetto figlio** e il contenitore **oggetto genitore**.

Ad esempio l'oggetto cane può appartenere all'oggetto canidi; a sua volta canidi può appartenere all'oggetto mammiferi; mammiferi può appartenere ad animali, ecc...

ESEMPLARI UNICI

Il sistema gerarchico oggetti-esemplari è molto comodo e potente ma non sempre è utile, perciò un esemplare può anche essere **unico** ovvero non ereditare alcuna caratteristica ma essere definito interamente durante la sua stessa creazione.

LE CARATTERISTICHE DI OGGETTI ED ESEMPLARI

Gli oggetti/esemplari possono avere le **caratteristiche** qui sotto riportate.

Queste caratteristiche coincidono con le loro **variabili locali integrate** nonché con gli **argomenti delle funzioni** a loro dedicate.

NOME: indica il nome dell'esemplare o dell'oggetto [default: *nome casuale alfanumerico*]
OGGETTO: indica l'oggetto a cui appartiene l'esemplare [default: *nessuno*]
GENITORE: indica l'oggetto genitore [default: *nessuno*]
X: indica la posizione sull'asse x [default: 0]
Y: indica la posizione sull'asse y [default: 0]
Z: indica la profondità, una profondità minore o uguale a 0 imposta l'oggetto al di sotto della griglia. [default: -1]
IMMAGINE: indica l'immagine dell'oggetto/esemplare. [default: *immagine predefinita*]
SCALA ASSE X: indica la scala dell'immagine sull'asse x [default: 1]
SCALA ASSE Y: indica la scala dell'immagine sull'asse y [default: 1]
TRASPARENZA: indica la trasparenza dell'immagine [default: 1]
COLORE: indica il colore (blend) dell'immagine [default: bianco]
VELOCITA: indica la velocità in px/step dell'oggetto/esemplare [default: 0]
DIREZIONE: indica la direzione in gradi dell'oggetto/esemplare [default: 0]
ROTAZIONE: indica la rotazione in gradi dell'immagine dell'oggetto/esemplare [default: 0]

Funzioni di base sugli oggetti

CREARE OGGETTI ED ESEMPLARI

Per creare oggetti bisogna utilizzare la funzione *crea un oggetto*:

```
crea un oggetto --> (NOME:) (GENITORE:) (X:) (Y:) (Z: ) (SCALA ASSE X:) (SCALA ASSE Y:) (IMMAGINE:) (TRASPARENZA:) (COLORE:)
                    (VELOCITA:)  (DIREZIONE:) (ROTAZIONE:)
```

Come per tutte le funzioni, è possibile omettere qualsiasi argomento lasciando intatto il valore di default, tuttavia omettendo il nome non sarà possibile utilizzare l'oggetto creato.

Esempio:

```
1  immagine_freccia=ottieni immagine → (IMMAGINE: "immagini/freccia.png")
2
3  crea un oggetto → (NOME: freccia)                    // Nome delloggetto
4                    (X: 20) (Y: 20)                    // Coordinate in cui verranno creati i suoi esemplari
5                    (DIREZIONE: -45) (ROTAZIONE: -45)  // Direzione di movimento e rotazione dellimmagine
6                    (VELOCITA: 3)                      // Velocità di movimento in pixel per step
7                    (IMMAGINE: immagine_freccia)       // Limmagine che rappresenta l'oggetto
```

Per creare esemplari bisogna utilizzare la funzione *crea un esemplare*:

```
crea un esemplare --> (NOME:) (OGGETTO:) (X:) (Y:) (Z: ) (SCALA ASSE X:) (SCALA ASSE Y:) (IMMAGINE:) (TRASPARENZA:) (COLORE:)
                      (VELOCITA:)  (DIREZIONE:) (ROTAZIONE:)
```

Creando un esemplare è possibile **aggiungere caratteristiche** e **modificare le caratteristiche ereditate**.

Esempi:

```
1  crea un esemplare → (OGGETTO: freccia)
2
3  crea un esemplare → (OGGETTO: freccia) (COLORE: verde)
4
5  crea un esemplare → (OGGETTO: freccia) (COLORE: viola) (VELOCITA: 10) (DIREZIONE: 180)
```

MODIFICARE UN ELEMENTO (OGGETTO/ESEMPLARE)

Per modificare un elemento (un oggetto o un esemplare) bisogna utilizzare la funzione *modifica un elemento*.

```
modifica un elemento --> (NOME:) (X:) (Y:) (Z: ) (SCALA ASSE X:) (SCALA ASSE Y:) (IMMAGINE:) (TRASPARENZA:) (COLORE:)
                         (VELOCITA:)  (DIREZIONE:) (ROTAZIONE:) (VELOCITA ANIMAZIONE:) (FOTOGRAMMA:)
```

Modificando un **esemplare** si modifica il **singolo esemplare** individuato.

Modificando un **oggetto** si modificano **tutti gli esemplari** di quell'oggetto ma **non il modello astratto**.

Modificando un **oggetto genitore** si modificano anche tutti gli **esemplari dell'oggetto figlio**.

Esempio:

```
1  modifica un elemento → (NOME: missile) (DIREZIONE: 45)
```

CREARE VARIABILI LOCALI

Ogni oggetto/esemplare oltre alle variaibli locali integrate può contenere delle **variabili locali personalizzate**.

Per dichiarare una variabile locale bisogna scriverla come se fosse un argomento della funzione *crea un oggetto* o *crea un esemplare,* ovvero nella forma (variabile: valore).

Esempio:

```
1 crea un oggetto ➔ (NOME: automobile) (benzina: 100)
```

Come nelle variabili globali per le variabili composte da più parole è necessario usare il simbolo "_". Esempio:

```
1 crea un oggetto ➔ (NOME: automobile) (colore_carrozzeria: blu)
```

N.B. le variabili locali possono anche essere scritte in maiuscolo ma questa pratica è sconsigliata, poiché compromette la coerenza sintattica e l'evidenziazione del codice.

Esempi:

```
1 crea un oggetto ➔ (NOME: automobile) (benzina: 100) //OK
2 crea un oggetto ➔ (NOME: automobile) (BENZINA: 100) //DA EVITARE!
```

LEGGERE UNA VARIABILE LOCALE

Per leggere una variabile locale ci sono due metodi.

Il più semplice è richiamarla all'interno di una funzione che evoca esplicitamente l'oggetto/esemplare.

Le funzioni in cui è possibile farlo sono: *crea un oggetto*, *crea un esemplare*, *modifica un elemento.*

La forma da utilizzare è (ETICHETTA: VARIABILE).

Esempio:

```
1 modifica un elemento ➔ (NOME: missile) (DIREZIONE: direzione) (ROTAZIONE: DIREZIONE)
2 //Langolo di rotazione del missile segue la sua stessa direzione
```

È anche possibile inserire la variabile in un'espressione. Esempio:

```
1 se tasto freccia destra è premuto allora modifica un elemento ➔ (NOME: giocatore) (X: X+3).
2 se tasto freccia sinistra è premuto allora modifica un elemento ➔ (NOME: giocatore) (X: X-3).
3 se tasto freccia su è premuto allora modifica un elemento ➔ (NOME: giocatore) (Y: Y-3).
4 se tasto freccia giu è premuto allora modifica un elemento ➔ (NOME: giocatore) (Y: Y+3).
5 //Il giocatore si sposta della sua coordinata attuale + 3
```

Il secondo metodo permette di leggere e utilizzare ovunque la variabile.

Utilizzando il costrutto "**del"** nella forma *<nome variabile> del <nome elemento>* è possibile leggere ed utilizzare una variabile locale all'interno di qualsiasi espressione.

Esempi:

```
1 INIZIA
2 crea un esemplare ➔ (NOME: cavaliere) (X: 300)
3 crea un esemplare ➔ (NOME: scudo)
4
5 CICLO CONTINUO
6 modifica un elemento ➔ (NOME: scudo) (X: X del cavaliere+50)
7 //posiziona lo scudo 50 pixel davanti al cavaliere
```

N.B. il costrutto "del" permette di leggere una variabile locale ma non di modificarla.
L'unico modo per modificare una variabile locale è utilizzando la funzione *modifca elemento*.

Ad esempio, non è possbile scrivere:

```
1 X del cavaliere = 500 //SBAGLIATO!!
```

La forma corretta da utilizzare è la seguente:

```
1 modifica un elemento → (NOME: cavaliere) (X: 500) //CORRETTO
```

Gli articoli partitivi **dell'**, **della** e **dello** possono essere usati in modo equivalente a **del**.

Ad esempio, queste scritture sono valide:

```
1 VELOCITA dellauto
2 ROTAZIONE della palla
3 peso dello zucchero
```

Funzioni avanzate sugli oggetti

ottieni risultato controllo collisione tra --> (ELEMENTO 1:) (ELEMENTO 2:)

Controlla se è avvenuta una collisione tra due elementi. La funzione restituisce 1 (*vero*) se la collisione è avvenuta o 0 (*falso*) se non si è verificata la collisione indicata.

Esempio:

```
CICLO CONTINUO
colpito = ottieni risultato controllo collisione tra ➔ (ELEMENTO 1: giocatore) (ELEMENTO 2: proiettile)
se colpito = vero allora diminuisci vita di 10.
```

ottieni risultato controllo se esistono esemplari di --> (ELEMENTO:)

Controlla se esistono esemplari di un oggetto o se esiste un esemplare con quel nome. Se l'esito del controllo è positivo la funzione restituisce 1 (*vero*) altrimenti restituisce 0 (*falso*).

Esempio:

```
CICLO CONTINUO
//Controlla se i nemici sono stati abbattuti: se si fai apparire tre nuovi nemici in una posizione casuale
nemico_presente = ottieni risultato controllo se esistono esemplari di ➔ (ELEMENTO: nemico)
se nemico_presente = falso
{
    ripeti per 3 volte:
    {
    x_casuale = ottieni un valore compreso tra questi ➔ (VALORE 1: 0) (VALORE 2: larghezza finestra)
    y_casuale = ottieni un valore compreso tra questi ➔ (VALORE 1: 0) (VALORE 2: altezza finestra)
    crea un esemplare ➔ (OGGETTO: nemico) (X: x_casuale) (Y: y_casuale)
    }
}
```

distruggi un esemplare --> (ESEMPLARE:)

Distrugge l'esemplare specificato.

Esempio:

```
CICLO CONTINUO
//Game Over!
se vita=0 allora distruggi un esemplare ➔ (ESEMPLARE: giocatore).
```

distruggi tutti gli esemplari di --> (OGGETTO:)

Distruggi tutti gli esemplari dell'oggetto specificato.

Esempio:

```
CICLO CONTINUO
//Distruggi tutte le porte bloccate nel gioco quando raggiungi la sala comandi
se uscite_sbloccate=vero  allora distruggi tutti gli esemplari di ➔ (ESEMPLARE: porta_bloccata).
```

ottieni nome dell'esemplare più vicino a --> (X:) (Y:) (OGGETTO:)

Individua l'esemplare dell'oggetto specificato che è più vicino alle coordinate specificate e restituisce il suo nome.

Esempio:

```
CICLO CONTINUO
//Lobbiettivo del missile è il nemico più vicino
obbiettivo = ottieni nome dell'esemplare più vicino a ➔ (X: X del missile) (Y: Y del missile) (OGGETTO: nemico)
```

ottieni nome dell'esemplare più lontano da --> (X:) (Y:) (OGGETTO:)

Individua l'esemplare dell'oggetto specificato che è più lontano alle coordinate specificate e restituisce il suo nome.

Esempio:

```
CICLO CONTINUO
//Lobbiettivo del missile è il nemico più lontano
obbiettivo = ottieni nome dell'esemplare più lontano da ➔ (X: X del missile) (Y: Y del missile) (OGGETTO: nemico)
```

ottieni numero esemplari esistenti di --> (OGGETTO:)

Controlla e restituisce il numero di esemplari presenti nella finestra dell'oggetto specificato.

Esempio:

```
CICLO CONTINUO
//Controlla se ci sono meno di 4 nemici nella finestra: se si fai apparire tre nuovi nemici una posizione casuale
nemici_presenti = ottieni numero esemplari esistenti di ➔ (OGGETTO: nemico)
se nemici_presenti <4
{
    ripeti per 3 volte:
    {
    x_casuale = ottieni un valore compreso tra questi ➔ (VALORE 1: 0) (VALORE 2: larghezza finestra)
    y_casuale = ottieni un valore compreso tra questi ➔ (VALORE 1: 0) (VALORE 2: altezza finestra)
    crea un esemplare ➔ (OGGETTO: nemico) (X: x_casuale) (Y: y_casuale)
    }
}
```

ottieni numero totale di esemplari esistenti

Controlla e restituisce il numero complessivo di tutti gli esemplari presenti nella finestra.

Esempio:

```
CICLO CONTINUO
// Controlla il numero complessivo di tutti gli esemplari presenti nella finestra
tot=ottieni numero totale di esemplari esistenti
totale_esemplari  = ottieni testo combinato ➔ (TESTO 1: ci sono  ) (TESTO 2: tot ) (TESTO 3: esemplari nella finestra)
disegna testo ➔ (TESTO: totale_esemplari) (X: 10) (Y: 10)
```

ottieni risultato controllo se elemento è stato cliccato --> (ELEMENTO:)

Controlla se l'elemento specificato è stato cliccato (click singolo). Se l'esito del controllo è positivo la funzione restituisce 1 (*vero*) altrimenti restituisce 0 (*falso*).

Esempio:

```
CICLO CONTINUO
//Se viene cliccata la palla (oggetto palla) diventa rossa
palla_cliccata = ottieni risultato controllo se elemento è stato cliccato ➔ (ELEMENTO: palla)
 se palla_cliccata = vero allora modifica elemento ➔ (OGGETTO: palla) (COLORE: rosso).
```

ottieni risultato controllo se elemento è cliccato --> (ELEMENTO:)

Controlla se l'elemento specificato è cliccato (click prolungato). Se l'esito del controllo è positivo la funzione restituisce 1 (*vero*) altrimenti restituisce 0 (*falso*).

Esempio:

```
CICLO CONTINUO
//Trascina la palla con il mouse
trascina = ottieni risultato controllo se elemento è cliccato ➔ (ELEMENTO: palla)
 se trascina = vero allora modifica un elemento ➔ (OGGETTO: palla) (X: x del mouse) (Y: y del mouse).
```

ottieni risultato controllo se il mouse è entrato in questo elemento--> (ELEMENTO:)

Controlla se il cursore del mouse è all'interno dell'area dell'elemento indicato. Se l'esito del controllo è positivo la funzione restituisce 1 (*vero*) altrimenti restituisce 0 (*falso*).

Esempio:

```
CICLO CONTINUO
//Se il mouse entra nella palla (oggetto palla), la palla diventa verde
mouse_dentro = ottieni risultato controllo se il mouse è entrato in questo elemento ➔ (ELEMENTO: palla)
se mouse_dentro = vero allora modifica elemento ➔ (OGGETTO: palla) (COLORE: verde).
```

ottieni risultato controllo se il mouse è uscito da questo elemento --> (ELEMENTO:)

Controlla se il cursore del mouse è uscito dell'area dell'elemento indicato. Se l'esito del controllo è positivo la funzione restituisce 1 (*vero*) altrimenti restituisce 0 (*falso*).

Esempio:

```
CICLO CONTINUO
//Se il mouse esce dalla palla (oggetto palla), la palla diventa gialla
mouse_dentro = ottieni risultato controllo se il mouse è uscito da questo elemento ➔ (ELEMENTO: palla)
se mouse_dentro = vero allora modifica elemento ➔ (OGGETTO: palla) (COLORE: giallo).
```

muovi un elemento verso un punto --> (ELEMENTO:) (X:) (Y:) (VELOCITA:)

Muovi l'elemento specificato verso le coordinate specificate alla velocità (pixel per step) specificata.

Esempio:

```
CICLO CONTINUO
//Muovi il giocatore verso il mouse quando il tasto sinistro del mouse è premuto
se tasto sinistro del mouse è premuto = vero
{
muovi un elemento verso un punto ➔ (ELEMENTO: giocatore) (X: x del mouse) (Y: y del mouse) (VELOCITA: 10)
}
```

ottieni risultato controllo se elemento è uscito dalla finestra --> (ELEMENTO:)

Controlla se la posizione dell'elemento indicato è oltre il bordo della finestra. Se l'esito del controllo è positivo la funzione restituisce 1 (*vero*) altrimenti restituisce 0 (*falso*).

Esempio:

```
CICLO CONTINUO
//riposiziona la palla al centro se esce dalla finestra
palla_fuori= ottieni risultato controllo se elemento è uscito dalla finestra ➔ (ELEMENTO: palla )
se palla_fuori = vero allora modifica un elemento ➔ (OGGETTO: palla) (X: larghezza finestra/2) (Y: altezza finestra/2).
```

ottieni risultato controllo se elemento ha toccato il bordo della finestra --> (ELEMENTO:)

Controlla se la posizione dell'elemento indicato coincide con il bordo della finestra. Se l'esito del controllo è positivo la funzione restituisce 1 (*vero*) altrimenti restituisce 0 (*falso*).

Esempio:

```
CICLO CONTINUO
//fai tornare indietro la palla quando tocca il bordo della finestra
collisione_con_bordo = ottieni risultato controllo se elemento è uscito dalla finestra ➔ (ELEMENTO: palla)
se collisione_con_bordo = vero allora modifica un elemento ➔ (OGGETTO: palla) (DIREZIONE: DIREZIONE-180).
```

trasporta elemento al lato opposto quando esce dalla finestra --> (ELEMENTO:)

trasporta l'elemento indicato al lato opposto della finestra quando la sua posizione è oltre il bordo della finestra.

Esempio:

```
CICLO CONTINUO
//teletrasporta la palla al lato opposto quando esce della finestra
trasporta elemento al lato opposto quando esce dalla finestra ➔ (ELEMENTO: palla)
```

ottieni risultato controllo se ci sono elementi in questo punto --> (X:) (Y:)

Controlla se è presente almeno un elemento in quella posizione. Se l'esito del controllo è positivo la funzione restituisce 1 (*vero*) altrimenti restituisce 0 (*falso*).

Esempio:

```
CICLO CONTINUO
//inserisci una moneta nel gioco in una posizione casuale
x_casuale = ottieni un valore compreso tra questi ➔ (VALORE 1: 0) (VALORE 2: larghezza finestra)
y_casuale = ottieni un valore compreso tra questi ➔ (VALORE 1: 0) (VALORE 2: altezza finestra)

//controlla se la posizione è libera
posizione_libera = ottieni risultato controllo se ci sono elementi in questo punto ➔ (X: x_casuale) (Y: y_casuale)
se posizione_libera = vero allora crea un esemplare ➔ (OGGETTO: moneta) (X: x_casuale) (Y: y_casuale).
```

distruggi elementi che si trovano in questo punto --> (X:) (Y:)

Distruggi tutti gli elementi le cui coordinate coincidono con il punto specificate.

Esempio:

```
CICLO CONTINUO
distruggi elementi che si trovano in questo punto ➔ (X: 500) (Y: 200)
```

ottieni distanza tra due elementi --> (ELEMENTO 1:) (ELEMENTO 2:)

Ottieni la distanza in pixel tra i due elementi indicati. Nel caso vengano indicati degli oggetti verranno considerati gli esemplari più vicini tra loro. La distanza tiene conto dell'immagine associata agli esemplari (maschera di collisione).

Esempio:

```
CICLO CONTINUO
//se il giocatore è distante meno di 300 pixel dal nemico, il nemico lo insegue
distanza = ottieni distanza tra due elementi ➔ (ELEMENTO 1: giocatore) (ELEMENTO 2: nemico)

se distanza<300
{
muovi un elemento verso un punto ➔ (ELEMENTO: nemico) (X: X del giocatore) (Y: Y del giocatore ) (VELOCITA: 10)
}
```

Funzioni sui percorsi

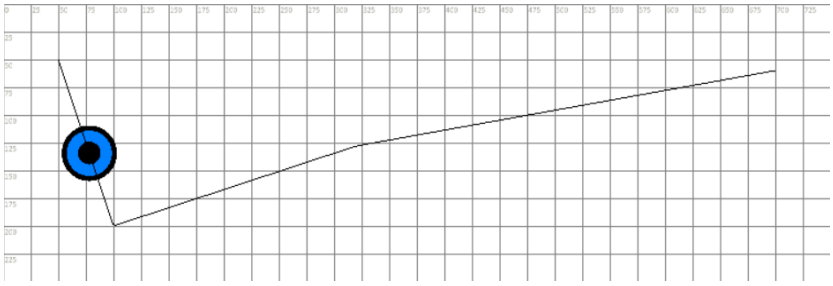
I percorsi sono degli insiemi di coordinate; sono delle risorse che vanno immagazzinate nelle variabili.

Grazie ai percorsi è possibile programmare movimenti complessi da far eseguire agli oggetti.

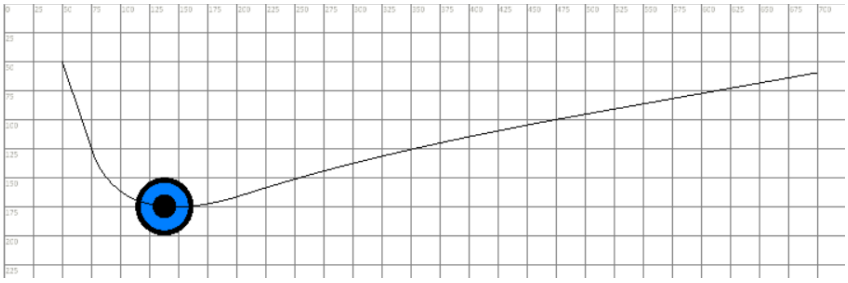
Per creare un percorso bisogna utilizzare la funzione *ottieni un percorso*.

percorso = ottieni un percorso --> (TIPO:) (CHIUSO:) (COORDINATE: x,y / x,y / …,…)

L'argomento **TIPO** indica il tipo di percorso: **dritto** o **curve**.

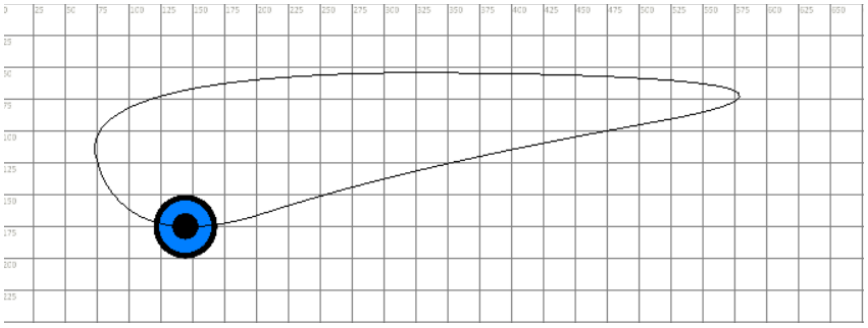


DRITTO

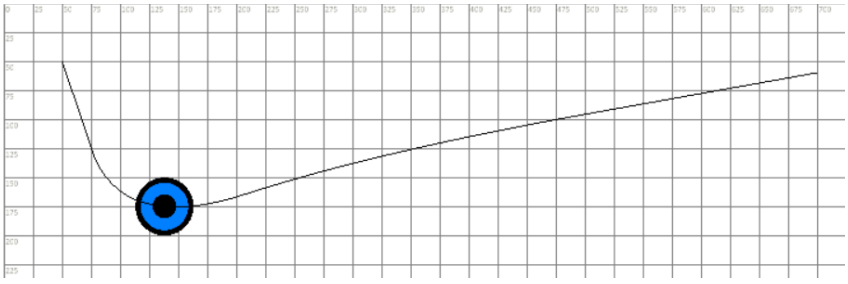


CURVE

L'argomento **CHIUSO** indica se il percorso è chiuso **(1, vero)** o aperto **(0, falso)**.



CHIUSO



APERTO

Le coordinate vanno specificate tramite la sintassi ***x,y/x,y/…***
Il simbolo "/"divide le coppie di ascisse e ordinate.

Esempio:

```
1 (COORDINATE: 50,50/100,200/320,128/700,60)
```

Per muovere un oggetto usando un percorso bisogna utilizzare la funzione *assegna un percorso da seguire a un elemento*.

assegna un percorso da seguire a un elemento --> (ELEMENTO:) (PERCORSO:) (VELOCITA:) (RELATIVO:) (QUANDO FINISCE:) (DISEGNA PERCORSO:)

l'argomento **ELEMENTO** definisce l'oggetto o l'esemplare a cui assegnare il percorso.
L'argomento **VELOCITA** assegna la velocità di percorrenza in pixel per step.
L'argomento **RELATIVO** (vero o falso) indica se il percorso va applicato a partire dalle coordinate dell'oggetto oppure se le coordinate da seguire sono assolute.
L'argomento **QUANDO FINISCE** permette di specificare all'oggetto che cosa fare quando raggiunge la fine del percorso, è possibile utilizzare diverse parole chiave:

- stai fermo
- torna al punto di partenza
- continua a muoverti in quella direzione
- torna indietro

L'argomento **DISEGNA PERCORSO** (vero o falso) permette di disegnare o meno il percorso assegnato all'oggetto

Esempio completo:

```
1  INIZIA
2  percorso_prova = ottieni un percorso ➔ (COORDINATE: 10,10 / 400,700 / 1000,500 ) (TIPO: curve) (CHIUSO: falso)
3  crea un esemplare ➔ (NOME: test)(OGGETTO: freccia ) (DIREZIONE: 0) (ROTAZIONE: DIREZIONE)
4
5  CICLO CONTINUO
6  se tasto invio è stato premuto = vero
7  {
8  assegna un percorso da seguire a un elemento ➔ (ELEMENTO: test) (VELOCITA: 1) (PERCORSO: percorso_prova)
9                                                 (QUANDO FINISCE: torna indietro) (DISEGNA PERCORSO: 1 )
10 }
```

Altre funzioni sui percorsi:

ferma movimento di un elemento --> (ELEMENTO:)

ottieni risultato controllo se questo elemento è arrivato alla fine del percorso assegnato --> (ELEMENTO:)

ottieni velocità percorrenza percorso --> (ELEMENTO:)

imposta velocità percorrenza percorso --> (ELEMENTO:) (VELOCITA:)

Funzioni crittografiche

In Atomic esistono due funzioni che permettono di cifare e decifrare messaggi utilizzando vari cifrari:

ottieni testo cifrato --> (TESTO:) (CIFRARIO:) (CHIAVE:)

ottieni testo decifrato --> (TESTO:) (CIFRARIO:) (CHIAVE:)

L'argomento **TESTO** è il testo che si intende cifrare o decifrare.

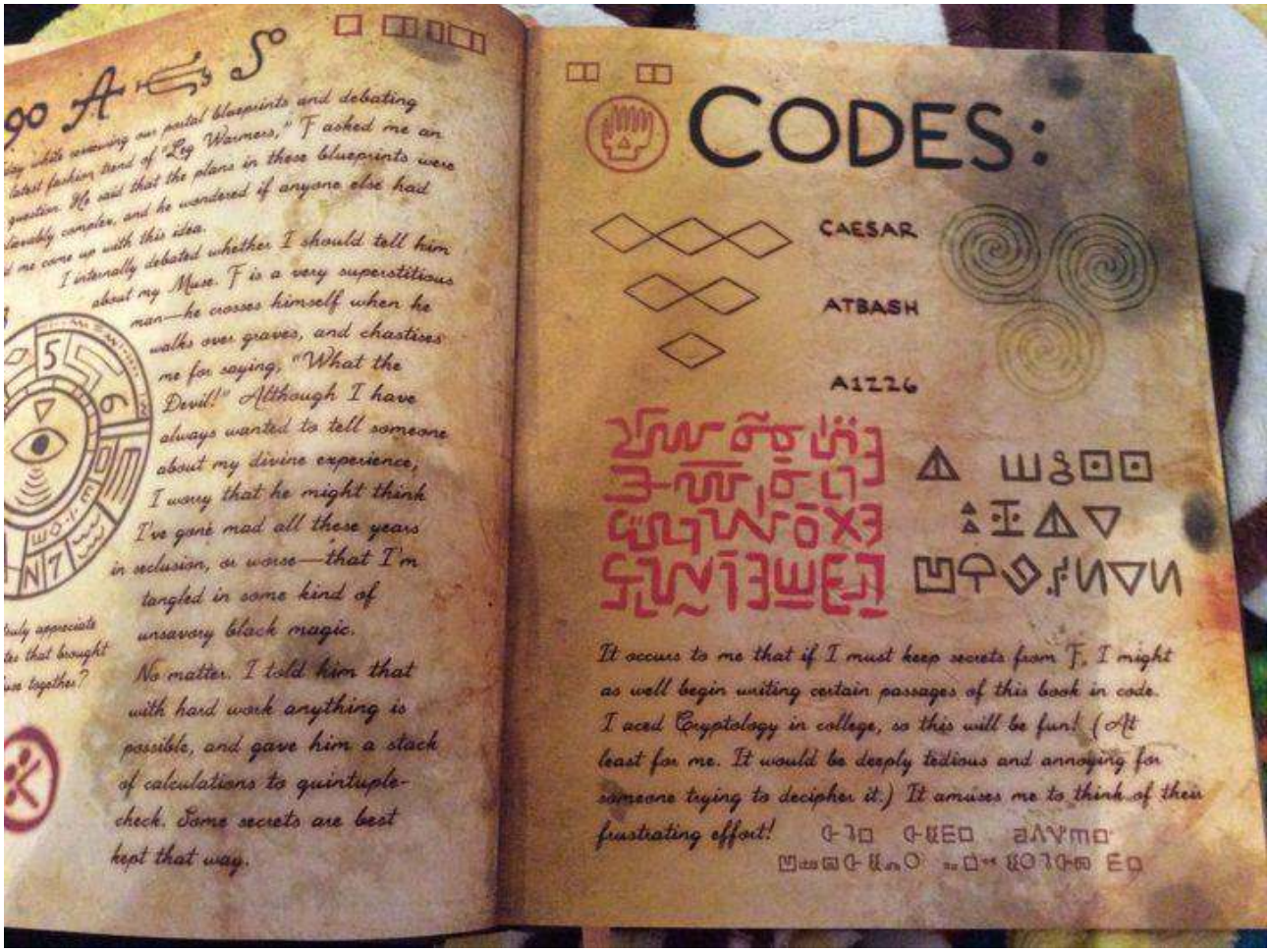
L'argomento **CIFARIO** è un testo che indica la tecnica crittografica utilizzata per cifrare o decifrare; i cifrari attualmente supportati sono: Atbash, Cesare, Sostituzione a sequenza, Parola chiave, Vigenere, Vernam.

L'argomento **CHIAVE** è il testo utilizzato per cifrare o decifrare, il testo necessario varia dal cifrario utilizzato.

Queste funzioni non sono *case sensitive*: è indifferente scrivere gli argomenti in maiuscolo o in maiuscolo, il funzionamento sarà identico e il risultato sarà sempre restituito in maiuscolo.

Di seguito sono illustrati i vari cifrai utilizzabili.

Tutti i cifrari proposti sono a chiave simmetrica e utilizzabili a scopo didattico anche senza utilizzare un pc.



“Atbash“

Atbash è uno dei cifrari più antichi del mondo dato che è stato utilizzato nella Bibbia. Il suo nome deriva dal nome di quattro lettere dell’alfabeto ebraico: le prime e le ultime due.

Il cifrario è molto semplice e non richiede una chiave, poiché la chiave utilizzabile è una sola: l’alfabeto al contrario.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Z	Y	X	W	V	U	T	S	R	Q	P	O	N	M	L	K	J	I	H	G	F	E	D	C	B	A

Si tratta di un semplicissimo cifrario a sostituzione monoalfabetica; la lettera A viene sostituita dalla Z, la B dalla Y, la F dalla U, ecc…

Esempio
Testo in chiaro: CIAO QUESTO MESSAGGIO E’ SEGRETO
Testo cifrato: XRZL JFVHGL NVHHZTTRL V HVTIVGL

```
1  INIZIA
2  messaggio = ottieni testo cifrato → (CIFRARIO: "Atbash")
3                                      (TESTO: "CIAO QUESTO MESSAGGIO E' SEGRETO")
4
5  crea casella di testo multilinea → (NOME: casella) (ETICHETTA: "messaggio decifrato") (TESTO: messaggio)
```

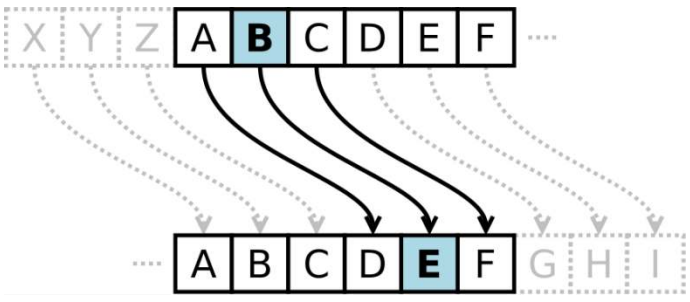
Metodi per violarlo
Una volta scoperto che il messaggio è cifrato con Atbash il cifario è automaticamente violato, poiché esiste una sola chiave.

“Cesare”

Il cifrario di Cesare è uno dei cifrari più antichi di cui si abbia traccia storica. Il suo nome deriva dal suo utilizzatore, il famoso Gaio Giulio Cesare che lo utilizzò durante le sue campagne militari in Gallia.

Il cifrario è molto semplice, la chiave è un numero da 1 a 26 che rappresenta il numero di lettere da traslare.
Qui sotto è riportato un cifrario con chiave 3.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W



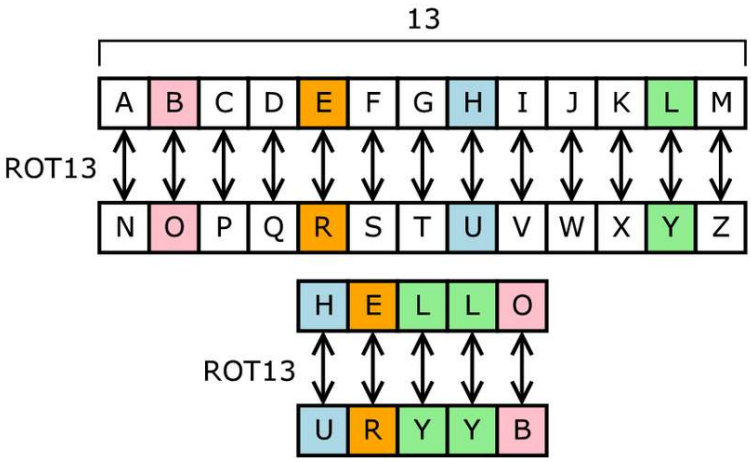
All’epoca di Cesare il cifrario era efficace a causa dell’alto tasso di analfabetismo e ignoranza in materia crittografica.
Il cifrario di Cesare, benché superato da più di mille anni, rimane importante poiché è la **base di vari cifrari più complessi**.

Esempio
Testo in chiaro: CIAO QUESTO MESSAGGIO E’ SEGRETO
Chiave: 3 Testo cifrato: FLDR TXHVZR PHVVDJJLR H VHJUHZR

```
INIZIA
messaggio = ottieni testo cifrato → (CIFRARIO: "Cesare") (CHIAVE: "3")
                                   (TESTO: "CIAO QUESTO MESSAGGIO E' SEGRETO")

crea casella di testo multilinea → (NOME: casella) (ETICHETTA: "messaggio decifrato") (TESTO: messaggio)
```

La versione del cifrario di Cesare a chiave 13 è chiamato **ROT13** e permette di utilizzare lo stesso algoritmo sia per la cifratura che per la decifratura: sopravvive ancora oggi per offuscare testi (in modo da non poter essere immediatamente leggibili) ad esempio nelle soluzioni dei giochi d'enigmistica.



Metodi per violarlo
Attacco a forza bruta: esistono solo 26 chiavi, provandole tutte si trova il messaggio in chiaro.

"Sostituzione a sequenza"

Il cifrario di Sostituzione a sequenza è un cifrario a sostituzione monoalfabetica molto elementare che consiste nel sostituire le lettere con una sequenza in ordine casuale delle lettere dell'alfabeto. Rientra nella categoria dei cifrari a sostituzione semplice.

La chiave non è altro che la sequenza di lettere utilizzata.

Qui sotto è riportato un cifrario con chiave "QWERTYUIOPASDFGHJKLZXCVBNM" ovvero le lettere secondo l'ordine della tastiera.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
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

Esempio
Testo in chiaro: CIAO QUESTO MESSAGGIO E' SEGRETO
Chiave: RABSZCTPDWEYOXMUNFQGVHIJKL
Testo cifrato: BDRM NVZQGM OZQQRTTDM Z QZTFZGM

```
INIZIA
messaggio = ottieni testo cifrato → (CIFRARIO: "Sostituzione a sequenza") (CHIAVE: "RABSZCTPDWEYOXMUNFQGVHIJKL")
                                   (TESTO: "CIAO QUESTO MESSAGGIO E' SEGRETO")

crea casella di testo multilinea → (NOME: casella) (ETICHETTA: "messaggio decifrato") (TESTO: messaggio)
```

Metodi per violarlo
Questo cifrario ha un'enormità di combinazioni di chiavi possibili rispetto a quello di Cesare (2^88,3 contro 26) ma nonostante un attacco a forza bruta risulta difficile da effettuare è comunque facile da violare tramite crittoanalisi. Al giorno d'oggi questo cifrario viene utilizzato solo come gioco d'enigmistica.

“Parola chiave”

Il cifrario a parola chiave è un cifrario a sostituzione monoalfabetica che utilizza sia la sostituzione che la traslazione. Rientra nella categoria dei cifrari a sostituzione semplice.

La chiave è una parola qualsiasi. La parola viene utilizzata come base della sostituzione, eliminando eventuali lettere ripetute; in seguito viene riportato il resto dell’alfabeto eliminando le lettere già utilizzate.

Qui sotto è riportato un cifrario con chiave “SEGRETO”. Notare la seconda E eliminata poiché già inserita.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
S	E	G	R	T	O	A	B	C	D	F	H	I	J	K	L	M	N	P	Q	U	V	W	X	Y	Z

Esempio
Testo in chiaro: CIAO QUESTO MESSAGGIO E’ SEGRETO
Chiave: SEGRETO
Testo cifrato: GCSK MUTPQK ITPPSAACK T PTANTQK

```
1  INIZIA
2  messaggio = ottieni testo cifrato → (CIFRARIO: "Cesare") (CHIAVE: "3")
3                                      (TESTO: "CIAO QUESTO MESSAGGIO E' SEGRETO")
4
5  crea casella di testo multilinea → (NOME: casella) (ETICHETTA: "messaggio decifrato") (TESTO: messaggio)
```

Metodi per violarlo
Leggermente più debole rispetto al cifrario a sostituzione a sequenza , l’attacco a forza bruta risulta comunque difficile da effettuare. Facilmente violabile tramite crittoanalisi, si può anche provare a forzarlo utilizzando un elenco di parole comuni.

“Vigenere”

Il cifrario di Vigenère è il più semplice dei cifrari polialfabetici. Si basa sull'uso di un versetto o di una parola per controllare l'alternanza degli alfabeti di sostituzione.

Pubblicato nel 1586, il cifrario di Blaise de Vigenère fu ritenuto per secoli inattaccabile. Una fama che è durata per molti anni anche dopo la scoperta del primo metodo di crittoanalisi da parte di Charles Babbage, e la successiva formalizzazione da parte del maggiore Friedrich Kasiski nel 1863.

Il metodo si può considerare una generalizzazione del cifrario di Cesare; invece di spostare sempre dello stesso numero di posti la lettera da cifrare, questa viene spostata di un numero di posti variabile ma ripetuto, determinato in base ad una parola chiave, da concordarsi tra mittente e destinatario, e da scrivere ripetutamente sotto il messaggio, carattere per carattere.

Le formule generiche che descrivono il cifrario sono le seguenti.

Per cifrare:

$$C_i \equiv T_i + K_i \pmod{m}$$

Per decifrare:

$$T_i \equiv C_i - K_i \pmod{m}$$

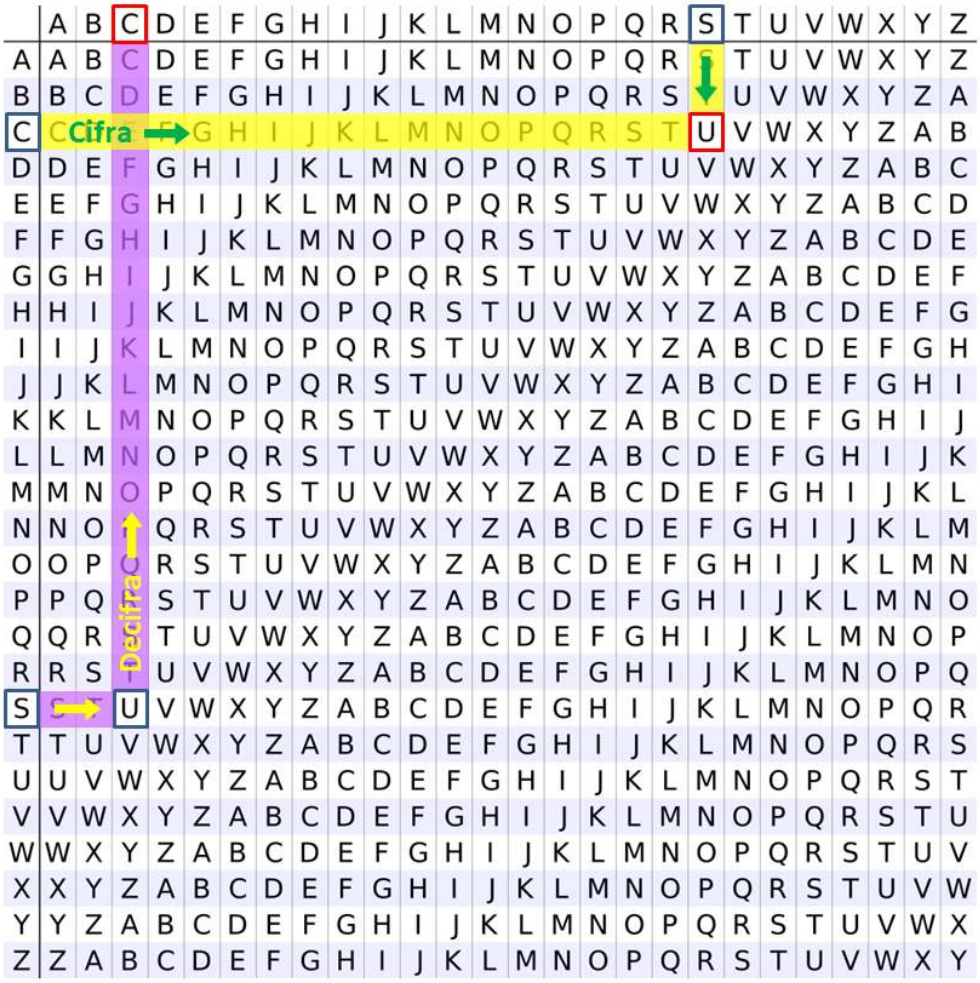
C_i - è il carattere cifrato

T_i - è il carattere del testo

K_i - è il carattere della chiave

m - è la lunghezza dell’alfabeto (26 nel nostro caso)

Per facilitare la cifratura e la decifratura Blaise de Vigenère propose l'uso di questa tavola:



Esempio

Testo in chiaro: CIAO QUESTO MESSAGGIO E' SEGRETO

Chiave: SEGRETO

Testo cifrato: UMGFUNSKXUDILGSKMZSXGWKXVXH

C	I	A	O	Q	U	E	S	T	O	M	E	S	S	A	G	G	I	O	E	S	E	G	R	E	...
S	E	G	R	E	T	O	S	E	G	R	E	T	O	S	E	G	R	E	T	O	S	E	G	R	...
U	M	G	F	U	N	S	K	X	U	D	I	L	G	S	K	M	Z	S	X	G	W	K	X	V	...

```
INIZIA
messaggio = ottieni testo cifrato → (CIFRARIO: "Vigenere") (CHIAVE: "Segreto")
                                    (TESTO: "CIAO QUESTO MESSAGGIO E' SEGRETO")

crea casella di testo multilinea → (NOME: casella) (ETICHETTA: "messaggio decifrato") (TESTO: messaggio)
```

Metodi per violarlo

Crittoanalisi, Metodo Kasiski

“Vernam”

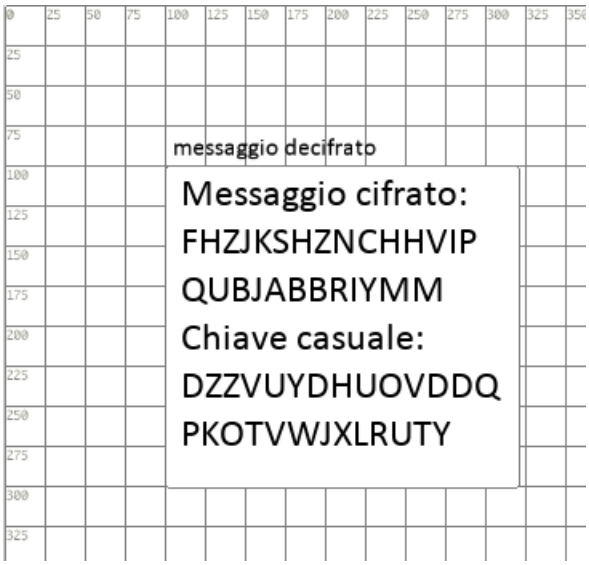
Il cifrario di Vernam è una versione migliorata del cifrario di Vigenère. È l’unico cifrario la cui inviolabilità è stata dimostrata matematicamente, per questo è l’unico cifrario ad essersi guadagno il titolo di “cifrario perfetto”. È stato utilizzato anche durante la guerra fredda per le comunicazioni tra Washinton e Mosca . Essenzialmente è un cifrario di Vigenère la cui chiave deve rispettare dei requisiti molto rigidi:

- La chiave deve essere generata casualmente
- La chiave deve essere lunga quanto il testo
- La chiave deve essere utilizzata una sola volta, per un solo messaggio

Nel caso di “Vernam” la funzione *ottieni messaggio cifrato* non richiede l’argomento CHIAVE, poiché la chiave pseudocasuale viene fornita assieme al testo cifrato.

Esempio:

```
1  INIZIA
2  messaggio = ottieni testo cifrato ➔ (CIFRARIO: "Vernam")
3                                      (TESTO: "CIAO QUESTO MESSAGGIO E' SEGRETO")
4
5  crea casella di testo multilinea ➔ (NOME: casella) (ETICHETTA: "messaggio decifrato") (TESTO: messaggio)
```



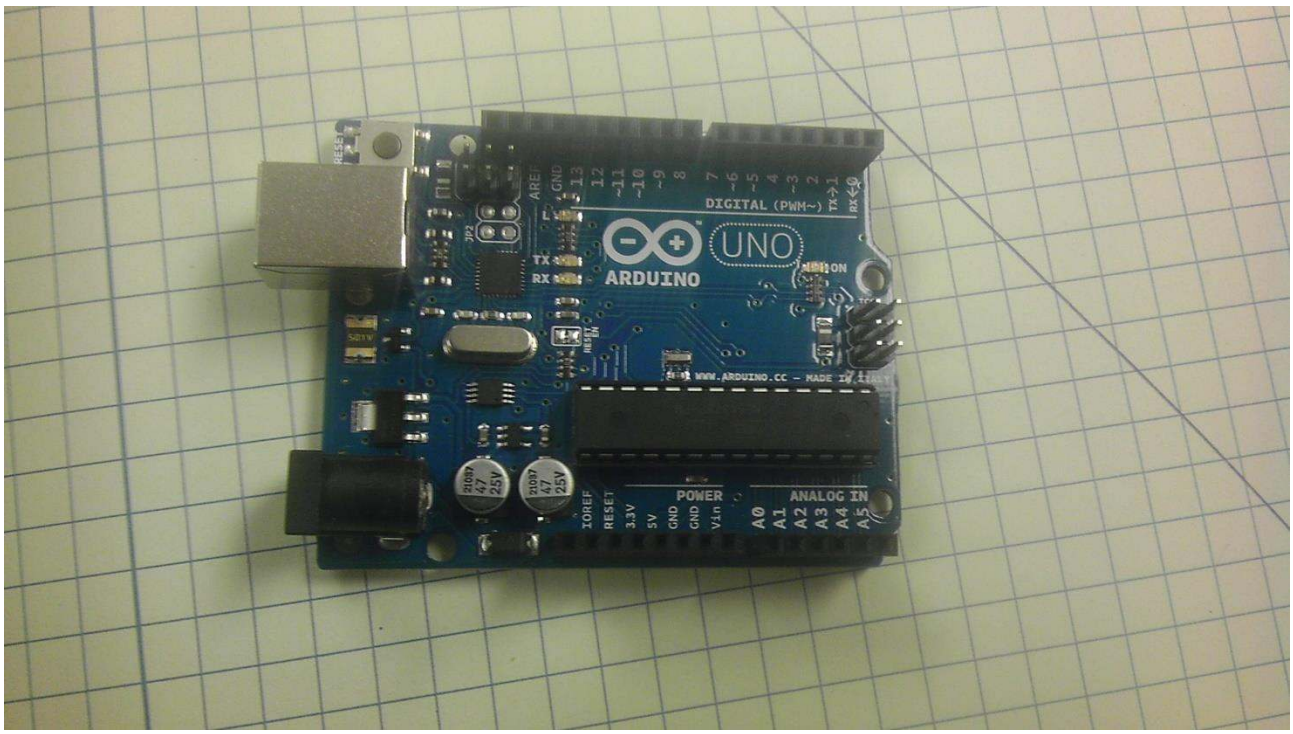
Testo in chiaro: CIAO QUESTO MESSAGGIO E’ SEGRETO
Chiave: DZZVUYDHUOVDDQPKOTVWJXLRUTY
Testo cifrato: FHZJKSHZNCHHVIPQUBJABBRIYMM

C	I	A	O	Q	U	E	S	T	O	M	E	S	S	A	G	G	I	O	E	S	E	G	R	E	…
D	Z	Z	V	U	Y	D	H	U	O	V	D	D	Q	P	K	O	T	V	W	J	X	L	R	U	…
F	H	Z	J	K	S	H	Z	N	C	H	H	V	I	P	Q	U	B	J	A	B	B	R	I	Y	…

Metodi per violarlo
Teoricamente impossibile da violare. Se la chiave è generata da un computer è possibile studiare l’algoritmo utilizzato per generare la chiave per tentare di violare il cifrario (metodo difficilmente applicabile agli algoritmi crittografici moderni). Questo poiché i computer possono simulare in modo verosimile la casualità ma non sono in grado di ottenere una casualità reale, perlomeno non senza interagire con fenomeni esterni.

Nonostante sia l’unico cifrario perfetto oggi è scarsamente utilizzato per la sua scomodità d’utilizzo (se il testo è molto lungo anche la chiave sarà altrettanto lunga) e per il fatto che non risolve uno dei più grandi problemi pratici della crittografia: la comunicazione a distanza della chiave. Tutti i cifrari visti fino ad ora funzionano tramite una chiave simmetrica; ovvero utilizzano la stessa chiave sia per cifrare che per decifrare. Il problema di utilizzare una chiave simmetrica è che chiunque entri in possesso della chiave può decifrare e cifrare nuovi messaggi; questo non solo vuol dire che qualsiasi malintenzionato entri in possesso della chiave può leggere i nostri messaggi ma può addirittura alterare il messaggio o scriverne un altro, fingendosi qualcun altro. Questi problemi dell’antichità sono stati risolti dalla crittografia asimmetrica (detta anche crittografia a coppia di chiavi o a chiave pubblica/privata).

Funzioni su Arduino



Arduino è un piccolo e semplicissimo computer utilizzato per realizzare velocemente progetti di elettronica.

Con Arduino si possono realizzare piccoli dispositivi come controllori di luci, di velocità per motori, sensori di luce, automatismi per il controllo della temperatura e dell'umidità e molti altri progetti che utilizzano sensori, attuatori e comunicazione con altri dispositivi. È abbinato ad un semplice ambiente di sviluppo integrato per la programmazione del microcontrollore.

Atomic può comunicare con Arduino (e schede cloni compatibili) tramite cavo USB, può quindi inviare e leggere dati. Per semplificare l'utilizzo di Arduino le funzioni sono essenziali e ridotte all'osso: molti aspetti sono gestiti automaticamente da Atomic.

È importante precisare che Atomic non sostituisce l'ambiente di sviluppo e il linguaggio con cui va programmato Arduino: semplicemente permette di interagire, e volendo di programmare, progetti Arduino creati appositamente a questo scopo.

Le funzioni riguardanti Arduino sono solo tre:

ottieni connessione con Arduino --> (PORTA:) (BAUD:)

La funzione *ottieni connessione con Arduino* Crea una connessione con Arduino utilizzando la **PORTA** ("COM 1","COM 2",ecc..) e il **BAUD** rate (il numero di simboli trasmessi in un secondo) specificati.

Se l'argomento BAUD non è specificato viene utilizzato il valore standard 9600.
Se l'argomento PORTA non viene specificato viene utilizzato il valore "COM1"
La funzione restituisce il valore "è connesso" o "connessione non riuscita"

invia testo ad Arduino --> (TESTO:)

La funzione *invia testo ad Arduino* invia il testo specificato ad Arduino.

ottieni testo da Arduino

La funzione *ottieni testo da Arduino* legge l'ultima riga inviata alla porta seriale.

Esempio completo

Qui sotto viene mostrato in ogni aspetto come creare un semplice progetto elettronico con Atomic e Arduino: si tratta di un programma che accende e spegne dei led tramite la tastiera del pc e che legge costantemente i messaggi inviati da Arduino sulla porta seriale (la stringa “Ciao Atomic!” più un numero che aumenta nel tempo). L’esempio mostra il codice per Atomic, lo sketch da caricare su Arduino (linguaggio Processing) e come collegare i componenti elettronici.

Codice per Atomic:

```
INIZIA
arduino = ottieni connessione con Arduino → (PORTA: "COM4") (BAUD: 9600)

CICLO CONTINUO
se arduino = "è connesso"
{
messaggio = ottieni testo da Arduino
disegna testo → (TESTO: messaggio)

   se tasto A è stato premuto = vero {invia testo ad Arduino → (TESTO: "Accendi il led")}
   se tasto S è stato premuto = vero {invia testo ad Arduino → (TESTO: "Spegni il led")}
}
```

Codice per Arduino:

```
int time=0;
String result;

void setup() {
  Serial.begin(9600);
  pinMode(13, OUTPUT);
  pinMode(12, OUTPUT);
}

void loop() {
  result = Serial.readString();
  if (result == "Accendi il led")  {
  digitalWrite(13, HIGH);
  digitalWrite(12, HIGH);
  }

  if (result == "Spegni il led")  {
  digitalWrite(13, LOW);
  digitalWrite(12, LOW);
  }

  time++;
  Serial.println("[inizio]Ciao Atomic!"+String(time)+"[fine]");
  delay(1);
}
```

Diagramma di collegamento:

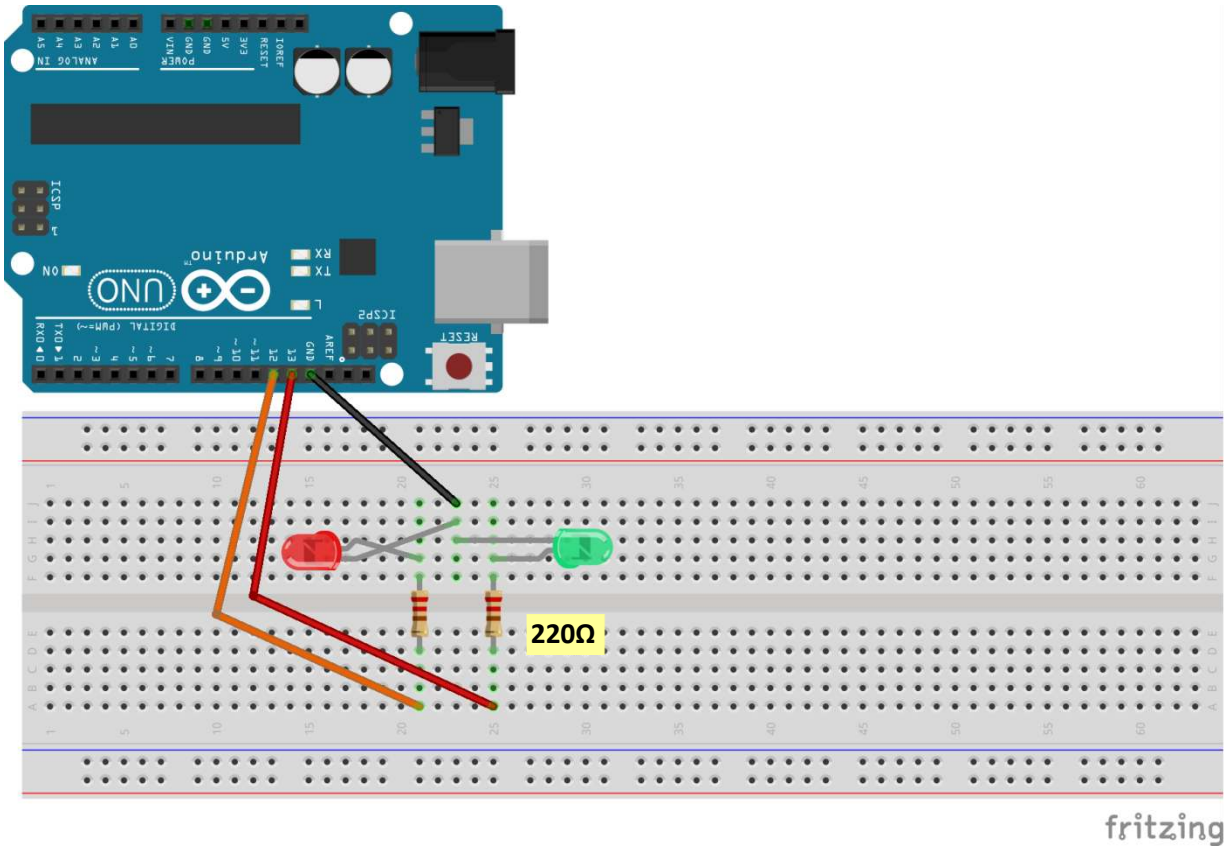
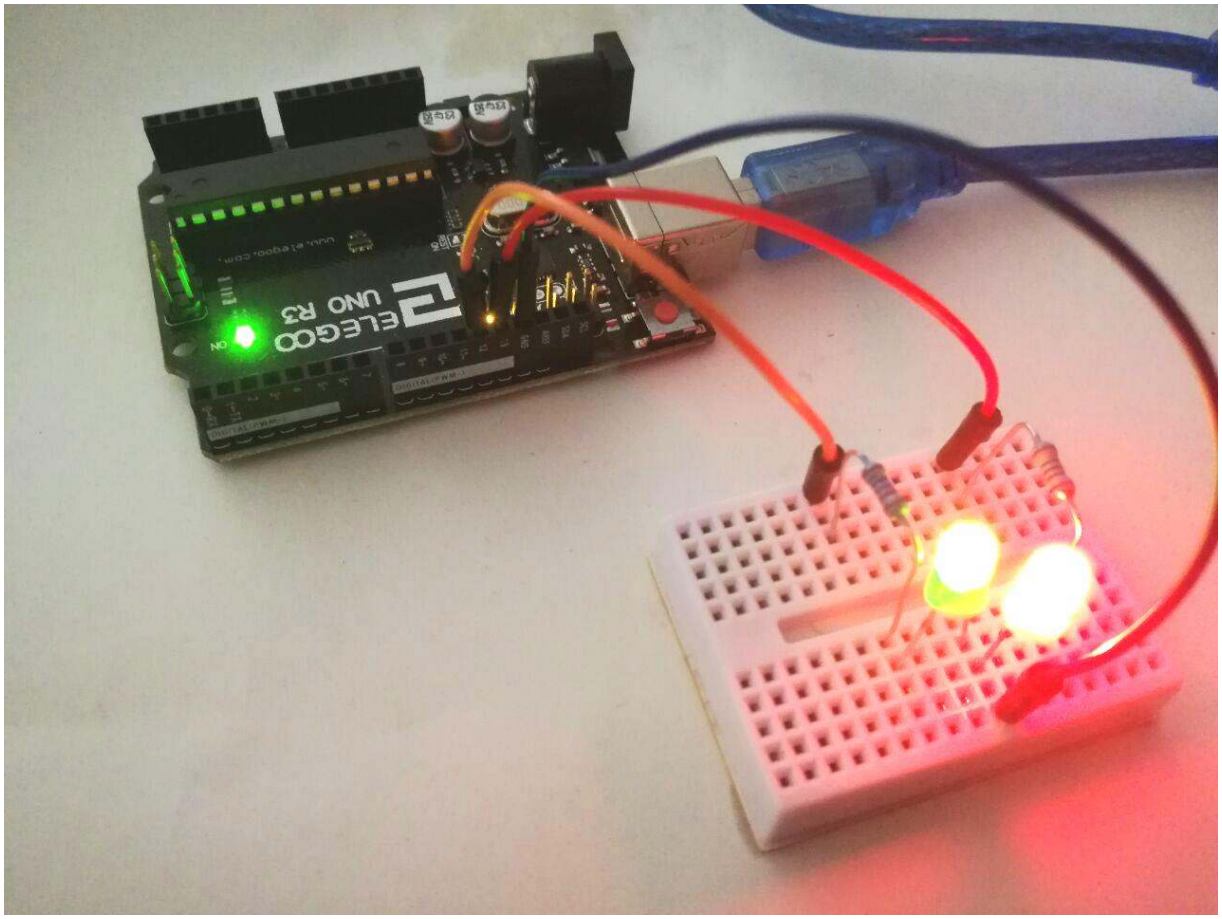


Foto d'esempio:



Note per lo sviluppo lato Arduino

Ricezione di messaggi

Arduino può leggere i valori inviati tramite la funzione *invia testo ad Arduino* usando le funzioni *Serial.parseInt()*, *Serial.parseFloat()* e *Serial.readString()* in base al tipo di dato che si vuole ottenere.

La funzione *Serial.readString()* è molto più lenta rispetto a *Serial.parseInt()*: è sconsigliato eseguire azioni in base alle stringhe ricevute se la temporizzazione è un elemento importante del progetto.

Invio di messaggi

Atomic può leggere messaggi inviati tramite la funzione *Serial.println()*, Il messaggio inviato da Arduino deve contenere i tag [inizio] e [fine] all'inizio e alla fine del messaggio, i messaggi che non rispettano questa sintassi vengono ignorati. Questi tag non appaiono nel testo ricevuto.

Per tutti i dettagli riguardanti la programmazione lato Arduino la documentazione ufficiale è disponibile all'indirizzo https://www.arduino.cc/reference/it/

Funzioni miscellanea

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

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

salva schermata

esci dal programma

riavvia il programma

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:

```
1 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:

```
1 aumenta orsetto_lavatore di 1
```

o per diminuirla, per esempio di 2:

```
1 diminuisci orsetto_lavatore di 2
```

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

```
1 aumenta orsetto_lavatore di 5*2
2 aumenta orsetto_lavatore di gatto
3 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:

```
1 //questo è un commento (scritta in verde). La funzione qui sotto disegna un cerchio
2 disegna cerchio → (RAGGIO: 200) (X: 300) (Y: 400)
3 //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 permettono 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 permettono 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:

```
1 /* tutto questo pezzo di codice verrà ignorato
2 disegna cerchio --> (RAGGIO: 200) (X: 300) (Y: 400)
3 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!

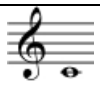
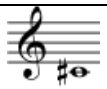
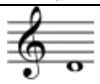
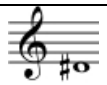
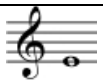
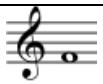
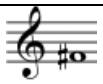
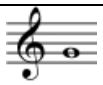
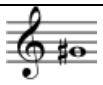
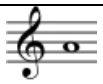
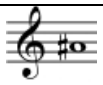
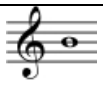
NOME	VALORE
è stato cliccato	“è stato cliccato”
non è cliccato	“non è cliccato”
è cliccato	“è cliccato”
al centro	“al centro”
a destra	“a destra”
a sinistra	“a sinistra”
in alto	“in alto”
in basso	“in basso”
radianti	“radianti”
gradi	“gradi”
è uguale a	“=“
è maggiore di	“>“
è minore di	“<“
è maggiore 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
suono nota piano	Do4 del pianoforte

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:

```
1 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:

```
1 se gatto > 0  allora disegna cerchio ➔ (RAGGIO: 200) (COLORE: blu).
2 se cane < 10 allora disegna cerchio ➔ (RAGGIO: 100) (COLORE: rosso).
3 se topo != cane allora disegna cerchio ➔ (RAGGIO: 150) (COLORE: giallo).
4 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:

```
1 se gatto è maggiore di 0 allora disegna cerchio ➔ (RAGGIO: 200) (COLORE: blu).
2 se cane è minore di 10 allora disegna cerchio ➔ (RAGGIO: 100) (COLORE: rosso).
3 se topo è diverso da cane allora disegna cerchio ➔ (RAGGIO: 150) (COLORE: giallo).
4 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:

```
1 se cane = 1 allora
2 disegna cerchio ➔ (RAGGIO: 100) (COLORE: rosso)
3 disegna rettangolo ➔ (LARGHEZZA: 200) (COLORE: verde)
4 topo = 300
5 .
```

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:

```
1 se cane = 1
2 {
3 disegna cerchio ➔ (RAGGIO: 100) (COLORE: rosso)
4 disegna rettangolo ➔ (LARGHEZZA: 200) (COLORE: verde)
5 topo = 300
6 }
```

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

```
1 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).
}
```

UTILIZZO DELLE VARIABILI TIMER

Come accennato nella sezione dedicata alle variabili integrate, in Atomic esistono delle variabili **timer** (timer 1, timer 2, … , timer 8)**.** Il loro funzionamento è molto semplice: ogni secondo queste variabili decrescono automaticamente di un'unità fino a raggiungere lo zero, simulando di fatto un conto alla rovescia.

Esempio:

```
INIZIA
timer 1 = 5

CICLO CONTINUO
se timer 1 = 0
{
suona → (SUONO: suono beep2)
timer 1 = 5
}
```

Questo codice fa emettere un suono al computer ogni 5 secondi.

È anche possibile specificare valori decimali per esprimere **decimi e centesimi di secondo**.

Esempio:

```
INIZIA
timer 1 = 5.23

CICLO CONTINUO
se timer 1 = 0
{
suona → (SUONO: suono beep 2)
timer 1 = 5.23
}
```

Questo codice fa emettere un suono al computer ogni 5.24 secondi, ovvero ogni 5 secondi, 2 decimi di secondo e 4 centesimi di secondo. **N.B. i timer lavorano in base agli fps e non in base all'orologio del computer**. Essendo Atomic volutamente limitato a 30 fps Il margine di precisione è di 0.03 secondi, ovvero 3 centesimi di secondo (1/30). Per questo motivo per ottenere una precisione assoluta i decimali devono essere un multiplo di 3.

L'utilizzo dei timer semplifica la gestione del tempo; qui sotto è riportato un esempio equivalente ma senza l'utilizzo dei timer.

Esempio senza timer:

```
INIZIA
tempo = 5*30

CICLO CONTINUO
diminuisci tempo di 1

se tempo = 0
{
suona → (SUONO: suono beep 2)
tempo = 5*30
}
```

Il tempo va moltiplicato per 30 (1 secondo = 30 passaggi) e va aggiunta una dichiarazione diminuisci per decrementare la variabile.

OPERATORI LOGICI

Gli operatori logici sono delle congiunzioni utili per legare insieme due o più condizioni fra loro; considerando come condizione una qualsiasi affermazione che può essere vera o falsa.

Esempi concettuali:

Luisa vorrebbe un fidanzato bello → bello = vero
Anna vorrebbe un fidanzato bello ma anche simpatico → bello = vero **e** simpatico = vero
Laura, che è meno esigente, vorrebbe un fidanzato bello o simpatico → bello = vero **o** simpatico = vero

Mentre per Luisa basta una sola variabile per esprimere la propria condizione, per quanto riguarda Anna e Laura le loro condizione sono più complesse: implicano l'utilizzo e la relazione tra due variabili. Nel caso di Anna la condizione si verifica solo se entrambe le variabili sono vere (lo vuole sia bello sia simpatico) mentre per quanto riguarda Laura (che si accontenta) basta che almeno una delle due variabili sia vera (lo vuole che sia almeno bello o che sia almeno simpatico, se poi è sia bello che simpatico tanto meglio!)

Questa logica è ciò che sta alla base dell'**algebra di Boole**. Questo ramo dell'algebra viene ampiamente utilizzato in informatica e in elettronica. Gli operatori di congiunzione visti sopra sono chiamati **operatori booleani**, **operatori logici** o **porte logiche**. Le espressioni contenenti gli operatori logici vengono chiamate **espressioni booleane**.

Atomic supporta parzialmente le espressioni booleane: 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.

Operatori logici unari

L'unico operatore unario supportato (tramite una funzione) è la negazione (**NOT**, NON).

NOME	VERO NOME	"SIMBOLO INFORMATICO"	SIMBOLO MATEMATICO
negazione	NOT	!	¬

Semplicemente la negazione inverte il vero con il falso e viceversa.
Per ottenere l'operatore **NOT**, è possibile usare questa funzione:

ottieni la negazione di --> (VALORE:)

Esempio:

```
1 gatto = vero
2 cane = 0
3 gatto2 = ottieni la negazione di → (VALORE: gatto)
4 cane2 = ottieni la negazione di → (VALORE: cane)
```

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

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

```
1 INIZIA
2 luce_accesa=0
3
4 CICLO CONTINUO
5 se tasto invio è stato premuto = vero allora luce_accesa = ottieni la 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).

Operatori logici binari

Questi sono gli operatori logici binari supportati:

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

Le espressioni booleane possono essere quindi utilizzate all'interno del costrutto se in questa forma:

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

Esempi:

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

"**e**" restituisce vero se entrambe le espressioni sono vere → (x ∧ y)=1 solo se (x=1 ∧ y=1)

"**o**" restituisce vero se almeno una delle espressioni è vera → (x ∨ y)=1 solo se (x=0 ∧ y=1) ∨ (x=1 ∧ y=0) ∨ (x=1 ∧ y=1)

"**o esclusivamente**" restituisce vero solo se una delle due espressioni è vera e l'altra falsa → (x⊕y)=1 solo se (x=0 ∧ y=1) ⊕ (x=1 ∧ y=0)

È possibile combinare più operatori logici all'interno di un'espressione booleana.

Esempio:

```
1 se coniglio = 1 e anatra = 0 o pollo=1 allora gatto = 4
```

I calcoli di verità vengono eseguiti da sinistra a destra (a differenza di altri linguaggi che seguono l'ordine di priorità).

Come per le espressioni aritmetiche **per modificare l'ordine di priorità dei calcoli è possibile utilizzare le parentesi tonde**.

Esempio:

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

N.B. Usando alcune tecniche è possibile riprodurre il funzionamento della maggior parte delle espressioni logiche senza utilizzare parentesi e proposizioni lunghe. Espressioni logiche troppo lunghe possono mandare in confusione anche i programmatori navigati; per questo, in ambito didattico, è utile "spezzettarle" in proposizioni più piccole.

Prendiamo ancora ad esempio questo codice:

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:

```
1  c1=0 //variabile contatore verità primo pezzo (coniglio, anatra)
2  c2=0 //variabile contatore verità secondo pezzo (cane, pollo, tacchino)
3
4  //c1
5  se coniglio=vero e anatra=falso allora c1=vero.
6
7  //c2
8  se cane=vero allora c2=vero.
9  se pollo=vero e tacchino=falso  allora  c2=vero.
10
11 //controllo finale di c1 e c2
12 se c1=vero o esclusivamente c2=vero allora gatto = vero.
```

Scritto in questo modo il codice è più ingombrante ma molto più schematico. In questo modo ogni passaggio è più 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 usando 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 (es. "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 il metodo sia chiaro e attinente al conteso (e ovviamente che funzioni!).

COSTRUTTO FINCHE

Il costrutto ***finche*** ha una sintassi molto simile al costrutto *se*:

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

Esempio:

```
1  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:

```
1  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 decine di righe di codice simili.

Un esempio pratico

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

```
1  INIZIA
2
3  CICLO CONTINUO
4  disegna cerchio ➔ (RAGGIO: 30) (COLORE: rosso) (X: 40)
5  disegna cerchio ➔ (RAGGIO: 30) (COLORE: rosso) (X: 40*2)
6  disegna cerchio ➔ (RAGGIO: 30) (COLORE: rosso) (X: 40*3)
7  disegna cerchio ➔ (RAGGIO: 30) (COLORE: rosso) (X: 40*4)
8  disegna cerchio ➔ (RAGGIO: 30) (COLORE: rosso) (X: 40*5)
9  disegna cerchio ➔ (RAGGIO: 30) (COLORE: rosso) (X: 40*6)
10 disegna cerchio ➔ (RAGGIO: 30) (COLORE: rosso) (X: 40*7)
11 disegna cerchio ➔ (RAGGIO: 30) (COLORE: rosso) (X: 40*8)
12 disegna cerchio ➔ (RAGGIO: 30) (COLORE: rosso) (X: 40*9)
13 disegna cerchio ➔ (RAGGIO: 30) (COLORE: rosso) (X: 40*10)
14 disegna cerchio ➔ (RAGGIO: 30) (COLORE: rosso) (X: 40*11)
15 disegna cerchio ➔ (RAGGIO: 30) (COLORE: rosso) (X: 40*12)
16 disegna cerchio ➔ (RAGGIO: 30) (COLORE: rosso) (X: 40*13)
17 disegna cerchio ➔ (RAGGIO: 30) (COLORE: rosso) (X: 40*14)
18 disegna cerchio ➔ (RAGGIO: 30) (COLORE: rosso) (X: 40*15)
19 disegna cerchio ➔ (RAGGIO: 30) (COLORE: rosso) (X: 40*16)
20 disegna cerchio ➔ (RAGGIO: 30) (COLORE: rosso) (X: 40*17)
21 disegna cerchio ➔ (RAGGIO: 30) (COLORE: rosso) (X: 40*18)
22 disegna cerchio ➔ (RAGGIO: 30) (COLORE: rosso) (X: 40*19)
23 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:

```
1  INIZIA
2  cerchi=0
3
4  CICLO CONTINUO
5  cerchi=1
6  finche cerchi < 20 allora disegna cerchio ➔ (RAGGIO: 30) (COLORE: rosso) (X: 40*cerchi)
7  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:

```
1  INIZIA
2  cerchi=0
3
4  CICLO CONTINUO
5  cerchi=1
6  finche cerchi < 20 allora disegna cerchio ➔ (RAGGIO: 5*cerchi) (COLORE: rosso) (X: 40*cerchi)
7  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!):

```
1  INIZIA
2  prova=150
3
4  CICLO CONTINUO
5  finche prova > 100 allora disegna cerchio ➔ (RAGGIO: 5*prova) (COLORE: rosso) (X: 40*prova)
6  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:

```
1  INIZIA
2  cerchi=0
3
4  CICLO CONTINUO
5  cerchi=1
6  ripeti per 20 volte : disegna cerchio ➔ (RAGGIO: 5*cerchi) (COLORE: rosso) (X: 40*cerchi)
7  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 " . "; esempio:

```
1  INIZIA
2  cerchi=0
3
4  CICLO CONTINUO
5  cerchi=1
6  ripeti per 20 volte
7  {
8  disegna cerchio ➔ (RAGGIO: 5*cerchi) (COLORE: rosso) (X: 40*cerchi)
9  aumenta cerchi di 1
10 }
```

TABELLE (ARRAYS)

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**.
Esempi:

Tabella monodimensionale (1D, vettore, lista)

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 utilizzate o 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

È anche possibile inserire tipi di dato diversi in una stessa colonna; esempio:

“Paolo”	43
21	“Maria”
“Giovanni”	10
323.54	56
“Silvia”	8,14

In Atomic non sono presenti 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).

Il sistema per riferirsi ai dati contenuti nelle celle di una tabella è molto semplice e si basa su RIGHE e COLONNE ordinate:

	COLONNA 1	COLONNA 2	COLONNA 3
RIGA 1	1	4324	523
RIGA 2	32	21	65
RIGA 3	2	76	25
RIGA 4	432432	6347	6
RIGA 5	32.1	654	3

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 dato da tabella --> (NOME:) (RIGA:) (COLONNA:)

Con questa funzione si può leggere il contenuto di una cella e memorizzarlo in una variabile
Se la tabella è monodimensionale si può omettere l'argomento COLONNA.

Modificare/dichiarare velocemente un'intera tabella

riempi tabella --> (NOME:) (COLONNA 1:) (COLONNA 2:) (COLONNA 3:) … (COLONNA 8:)

con questa funzione è possibile riempire tutte le celle di una tabella specificando una stringa di testo e usando il simbolo "|" per separare i valori di una colonna .

Esempio:

```
1  INIZIA
2  crea tabella → (NOME: esempio)
3  riempi tabella → (NOME: esempio) (COLONNA 1: "Mario|Alberto|Francesco|Luisa|Chiara|Giovanni|Luca|")
4                                   (COLONNA 2: "235|54|64.5|120|26|72|142|")
```

il risultato sarà questa tabella:

ESEMPIO	
Mario	235
Alberto	54
Francesco	64.5
Luisa	120
Chiara	26
Giovanni	72
Luca	142

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 una tabella

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

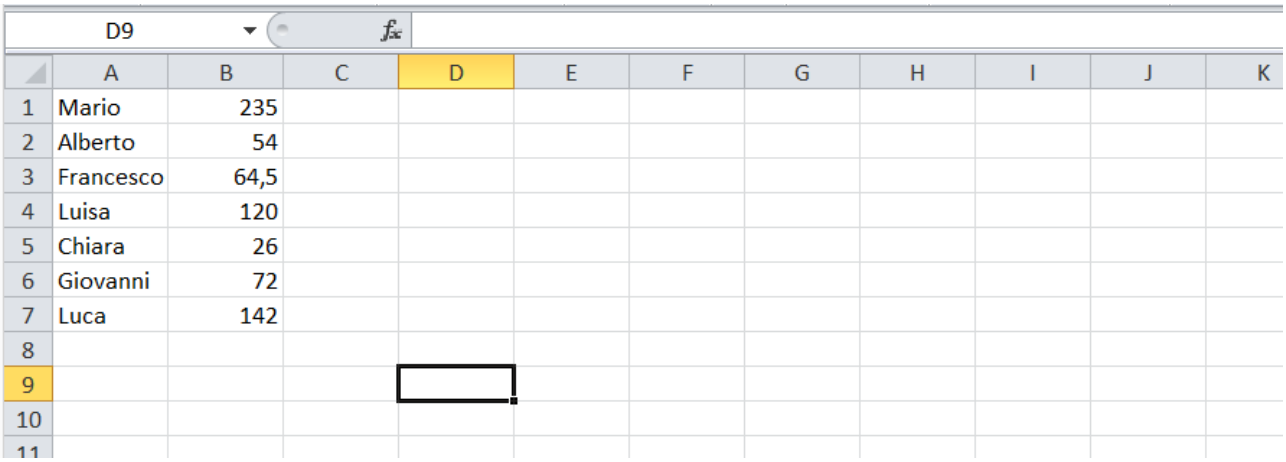
è possibile esportare tabelle in formato . csv (compatibili con Microsoft Excel e altri programmi simili).

Esempio:

```
1  INIZIA
2  crea tabella ➔ (NOME: esempio)
3  riempi tabella ➔ (NOME: esempio) (COLONNA 1: "Mario|Alberto|Francesco|Luisa|Chiara|Giovanni|Luca|")
4                                   (COLONNA 2: "235|54|64,5|120|26|72|142|")
5
6  esporta tabella ➔ (NOME: esempio) (INDIRIZZO: "tabella.csv")
```

tabella.csv

Questo esempio crea il file *tabella.csv* nella cartella *%LOCALAPPDATA%/Atomic*.
Il file generato può essere aperto e modificato con Microsoft Excel.



D9

	A	B	C	D	E	F	G	H	I	J	K
1	Mario	235									
2	Alberto	54									
3	Francesco	64,5									
4	Luisa	120									
5	Chiara	26									
6	Giovanni	72									
7	Luca	142									
8											
9											
10											
11											

Funzioni avanzate sulle tabelle: ! Queste funzioni non sono ancora state implementate.

ottieni il valore più basso tra questi valori in tabella --> (NOME:)(COLONNA:)
ottieni il valore più alto tra questi valori in tabella --> (NOME:)(COLONNA:)
ottieni la riga in cui si trova la prima occorrenza di questo valore in tabella --> (NOME:)(VALORE:)
ottieni la colonna in cui si trova la prima occorrenza di questo valore in tabella --> (NOME:)(VALORE:)
mischia a caso le celle di questa tabella --> (NOME:)
ordina in ordine ascendente le righe della tabella in base a una colonna --> (NOME:)(COLONNA:)
ordina in ordine decrescente le righe della tabella in base a una colonna --> (NOME:)(COLONNA:)
ordina in ordine ascendente le righe della tabella in base a una colonna --> (NOME:)(COLONNA:)
ordina in ordine alfabetico le righe della tabella in base a una colonna --> (NOME:)(COLONNA:)
ordina in ordine alfabetico inverso le righe della tabella in base a una colonna --> (NOME:)(COLONNA:)
copia tabella --> (NOME:) (NOME NUOVA TABELLA:)

DEFINIRE LE PROPRIE FUNZIONI

È possibile definire nuove funzioni usando il costrutto **definisci funzione**. La sintassi è la seguente:

definisci funzione "nome funzione"
{
… codice …
}

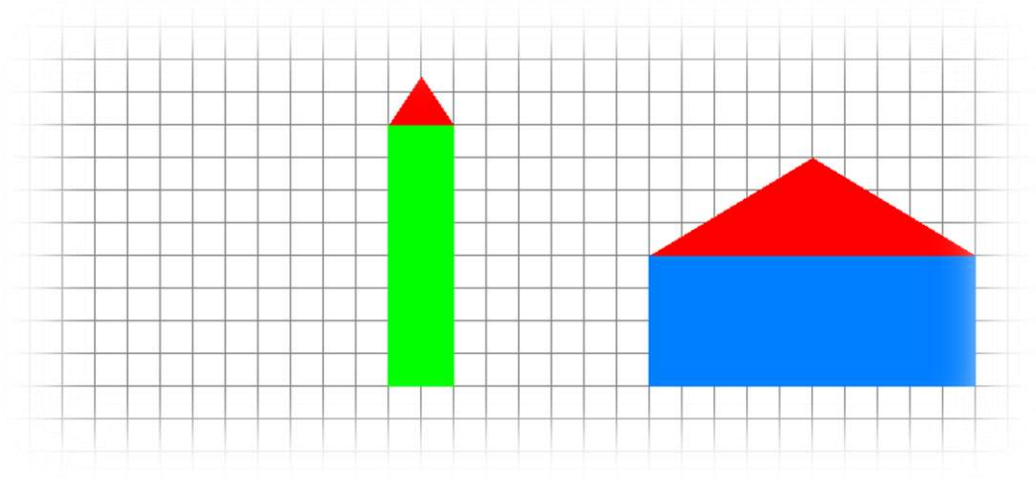
Definendo nuove funzioni è possibile **sostituire intere parti di codice con poche parole o una frase**. Questo rende più veloce la scrittura e rende il codice più snello e leggibile, soprattutto se contiene **parti ridondanti** la cui scrittura non può essere evitata tramite *cicli ripeti per* e *finche*.

Il nome della funzione deve essere scritto tra le virgolette (come stringa) e può quindi contenere spazi vuoti.
Tra le graffe è possibile inserire qualsiasi funzione, espressione e costrutto.
All'interno delle graffe è anche possibile **definire degli argomenti** tramite delle etichette (stringhe) tra i simboli "< >".
Queste etichette possono corrispondere agli argomenti già integrati in Atomic oppure essere diverse.

Esempio:

```
definisci funzione "disegna casa"
{
disegna rettangolo → (X: <"X">) (Y: <"Y">-<"ALTEZZA">) (BASE: <"LARGHEZZA">) (ALTEZZA: <"ALTEZZA">) (COLORE: <"COLORE">)
disegna triangolo → (X 1: <"X">) (Y 1: <"Y">-<"ALTEZZA">) (X 2: <"X">+<"LARGHEZZA">) (Y 2: <"Y">-<"ALTEZZA">)
(X 3: <"X">+<"LARGHEZZA">/2) (Y 3: <"Y">-<"ALTEZZA">-100/[<"ALTEZZA">/75]) (COLORE: rosso)
}

disegna casa → (LARGHEZZA: 250) (ALTEZZA: 100) (X: 500) (Y: 300) (COLORE: azzurro)
disegna casa → (LARGHEZZA: 50) (ALTEZZA: 200) (X: 300) (Y: 300) (COLORE: verde)
```



Gli argomenti definiti all'interno della funzione verranno sostituiti con valori reali nel momento in cui la funzione verrà richiamata all'interno del codice.
Ad esempio se si scrive **disegna casa --> (X: 500)** tutti i punti **<"X">** verranno modificati automaticamente in **500**:

```
disegna rettangolo --> (X: <"X">) (Y: <"Y">-<"ALTEZZA">) (BASE: <"LARGHEZZA">) (ALTEZZA: <"ALTEZZA">) (COLORE: <"COLORE">)
disegna triangolo -->  (X 1: <"X">) (Y 1: <"Y">-<"ALTEZZA">) (X 2: <"X">+<"LARGHEZZA">) (Y 2: <"Y">-<"ALTEZZA">)
(X 3:  <"X">+<"LARGHEZZA">/2)
(Y 3: <"Y">-<"ALTEZZA">-100/[<"ALTEZZA">/75]) (COLORE: rosso)
```

⇩

```
disegna rettangolo --> (X: 500) (Y: <"Y">-<"ALTEZZA">) (BASE: <"LARGHEZZA">) (ALTEZZA: <"ALTEZZA">) (COLORE: <"COLORE">)
disegna triangolo -->  (X 1: 500) (Y 1: <"Y">-<"ALTEZZA">) (X 2: 500+<"LARGHEZZA">) (Y 2: <"Y">-<"ALTEZZA">)
(X 3:  500+<"LARGHEZZA">/2)
(Y 3: <"Y">-<"ALTEZZA">-100/[<"ALTEZZA">/75]) (COLORE: rosso)
```

Questo vale per tutti gli argomenti specificati.

DEFINIRE FUNZIONI CHE RESTITUISCONO UN VALORE

È anche possibile definire funzioni che **restituiscono un valore**, ovvero che permettono di **ottenere un valore**:

definisci funzione "ottieni … nome funzione"
{
… codice …

con questa funzione ottieni …
}

Le uniche due differenze da una funzione normale sono:

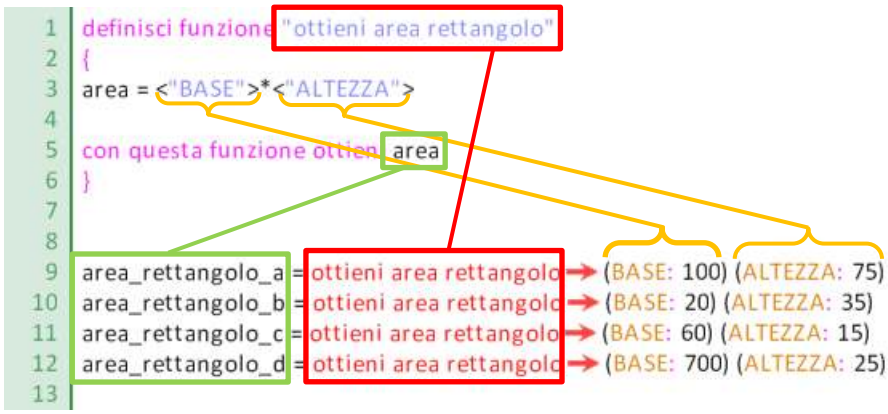
- il nome della funzione deve iniziare con "**ottieni**"
- il codice tra le graffe deve contenere il costrutto "**con questa funzione ottieni <espressione>"**

Il costrutto "**con questa funzione ottieni"** indica il valore (ricavato da una variabile o da un'espressione) che si otterrà utilizzando quella funzione.

Esempio:

```
1  definisci funzione "ottieni area rettangolo"
2  {
3  area = <"BASE">*<"ALTEZZA">
4  con questa funzione ottieni area
5  }
6
7  area_rettangolo_a = ottieni area rettangolo ➔ (BASE: 100) (ALTEZZA: 75)
8  area_rettangolo_b = ottieni area rettangolo ➔ (BASE: 20) (ALTEZZA: 35)
9  area_rettangolo_c = ottieni area rettangolo ➔ (BASE: 60) (ALTEZZA: 15)
10 area_rettangolo_d = ottieni area rettangolo ➔ (BASE: 700) (ALTEZZA: 25)
11
12 disegna testo ➔ (TESTO: area_rettangolo_a) (Y: 20)
13 disegna testo ➔ (TESTO: area_rettangolo_b) (Y: 40)
14 disegna testo ➔ (TESTO: area_rettangolo_c) (Y: 60)
15 disegna testo ➔ (TESTO: area_rettangolo_d) (Y: 80)
```

Schema di funzionamento:



Il costrutto "con questa funzione ottieni" può anche contenere direttamente gli argomenti specificati per la funzione.
Ad esempio il codice soprariportato può anche essere abbreviato in questo modo:

```
1  definisci funzione "ottieni_area_rettangolo" { con questa funzione ottieni <"BASE">*<"ALTEZZA"> }
```

Attenzione ai nomi che diamo alle funzioni: se da un lato definire nuove funzioni rende il codice meno ingombrante, 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 scegliere un nome chiaro per ogni funzione. Ad esempio, se creiamo una funzione che disegna una stella un buon nome può essere "disegna stella", un pessimo nome è un'abbreviazione tipo "distel" o "ds".

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 del programma e la correzione degli errori.

INSERIRE SUGGERIEMENTI PER LE PROPRIE FUNZIONI

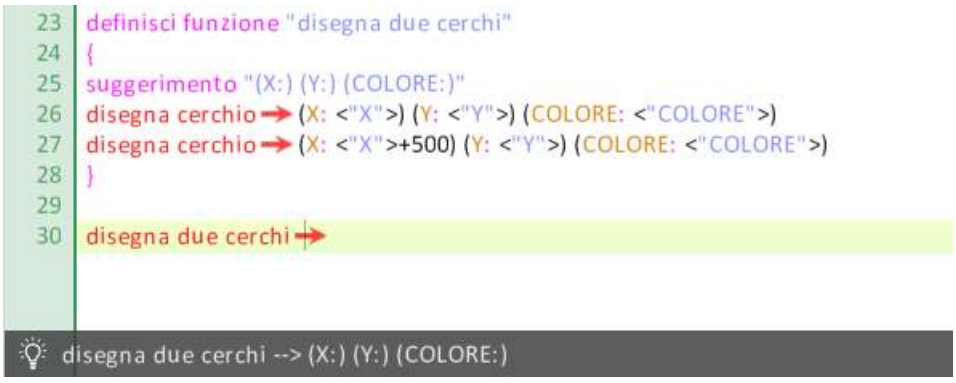
È possibile inserire dei suggerimenti per le proprie funzioni. Questi suggerimenti verranno visualizzati nella barra in basso dell'editor esattamente come per le funzioni integrate. La sintassi da utilizzare è la seguente:

Suggerimento "testo del suggerimento"

Il costrutto suggerimento va utilizzato all'interno del blocco (delimitato dalle parentesi graffe) che definisce la funzione:

definisci funzione "nome funzione"
{
Suggerimento "testo del suggerimento"
… codice …
}

Esempio:



Il testo del suggerimento comparirà dopo il simbolo -->.
È possibile definire qualsiasi testo per il suggerimento ma è consigliato attenersi a questa sintassi basilare:

(ARGOMENTO 1:) (ARGOMENTO 2:) (ARGOMENTO 3:) …

Inserendo, se necessario, altre informazioni tra i simboli " :) ".

Specificare il suggerimento per una funzione personalizzata **è fondamentale per far si che l'utilizzatore capisca il suo utilizzo**, soprattutto se la definizione della funzione è inclusa dall'esterno.

INCLUDERE CODICE ESTERNO

Per includere del codice (Atomic) esterno in qualsiasi punto basta utilizzare il costrutto **includi**:

includi "nome del file.txt"

il file deve trovarsi in **%LOCALAPPDATA%/Atomic/.**

È possibile specificare file che si trovano in sottocartelle di %LOCALAPPDATA%/Atomic/. Esempio:

```
1 includi "codici/esempi_facili/esempio.txt"
```

All'interno del codice esterno è possibile definire funzioni che potranno poi essere usate nel codice invocante.

Definire le proprie funzioni e includerle tramite file esterni permette di estendere l'operatività di Atomic.
In sintesi questo metodo di lavoro "a livelli" permette di:

- realizzare lavori più vasti e articolati
- suddividere il codice in modo chiaro in piccole parti ben distinte
- poter modificare progetti complessi usando funzioni molto specifiche e chiare definite dall'autore del progetto
- poter modificare progetti complessi usando poche righe di codice
- poter riciclare grandi quantità di codice (fino a creare delle proprie librerie)

Il grande vantaggio in ambito didattico è la possibilità di offrire l'accesso a progetti complessi tramite un interfaccia semplice e "sicura" agli studenti meno esperti. Allo stesso tempo gli studenti più avanzati hanno la possibilità di modificare il codice che si trova "sotto il cofano" dei progetti. **Inoltre l'evidenziazione delle funzioni personalizzate e i suggerimenti funzionano anche quando il codice che definisce le funzioni è incluso dall'esterno.**

Attenzione agli eventi!
Il costrutto includi funziona in modo molto semplice: prende il codice contenuto nel file specificato e lo "inietta" nel punto in cui è presente il costrutto. **Se il file iniettato contiene già gli eventi INIZIA e CICLO CONTINUO, e a sua volta il codice su cui stiamo lavorando li contiene, il codice finale verrà diviso in più eventi, creando quindi un malfunzionamento.**

DEBUG

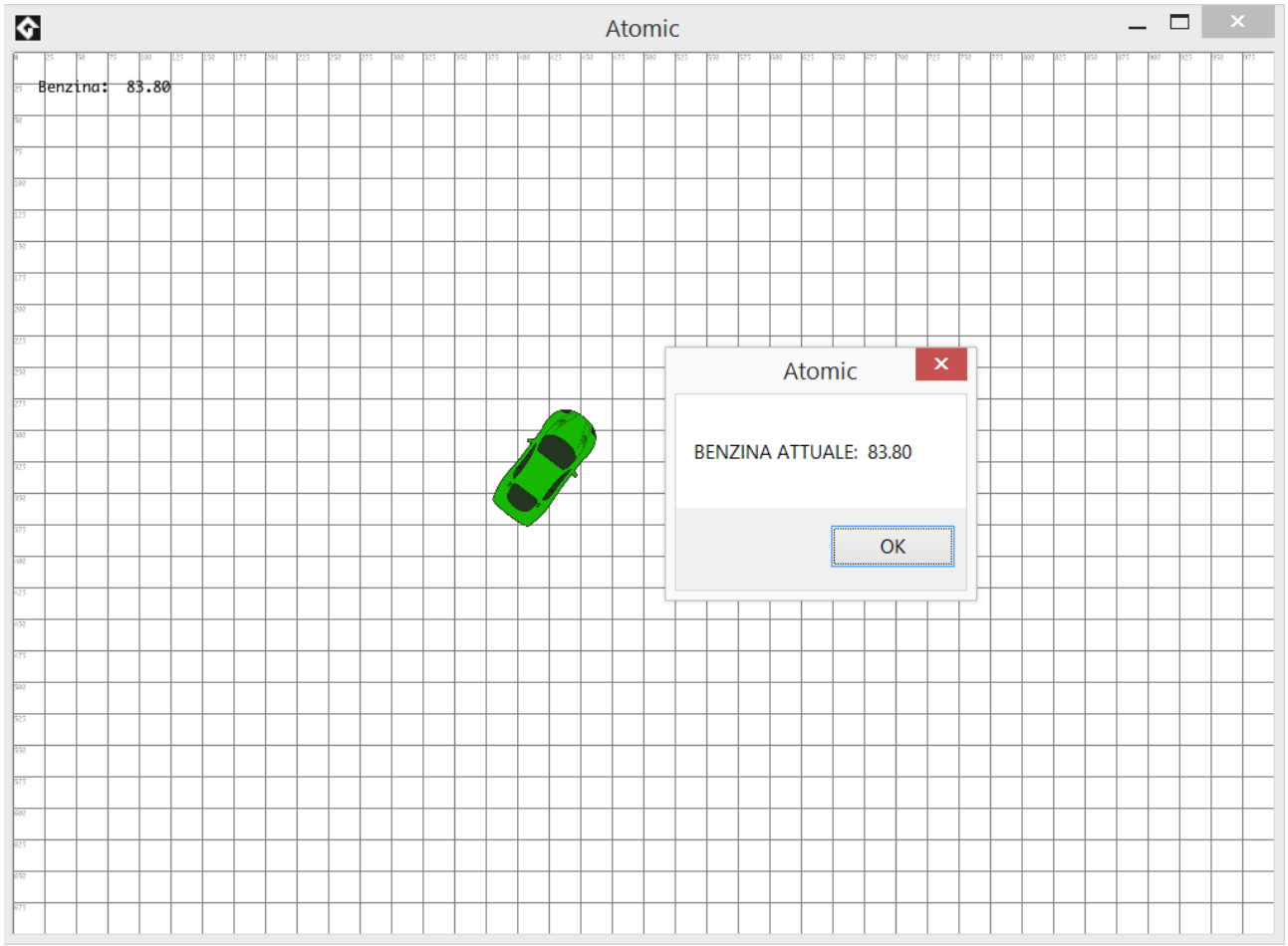
Tramite il costrutto **debug** è possibile fermare momentaneamente l'esecuzione del programma e mostrare un qualsiasi messaggio, anche inerente al codice, ad esempio mostrando il valore di una variabile o il risultato di un'espressione in quel determinato momento. È possibile utilizzare il costrutto debug anche all'interno di cicli finche e ripeti per.

La sintassi è molto semplice:

debug "stringa"

Esempio:

```
CICLO CONTINUO
//Accellerazione, freno e retromarcia
se benzina dell'auto è maggiore di 0
{
   se tasto freccia su è premuto = vero {
   modifica un elemento → (NOME: auto)  (VELOCITA: VELOCITA+accelerazione) (benzina: benzina-0.1)
   }
   se tasto freccia giu è premuto = vero{
   debug "BENZINA ATTUALE: <benzina dell'auto>"
   modifica un elemento → (NOME: auto)  (VELOCITA: VELOCITA-accelerazione*3) (benzina: benzina-0.1)
   }
}
```



IMPORTARE FUNZIONI ESTERNE TRAMITE DLL

È possibile importare nuove funzioni in Atomic tramite la funzione “importa funzione esterna”.

Questa funzione permette di utilizzare delle **dynamic-link library (DLL)** per estendere l’operatività di Atomic, trasformando funzioni e programmi scritti in linguaggi “avanzati” in funzioni Atomic facili da utilizzare.

La sintassi da utilizzare è la seguente:

```
importa funzione esterna --> (INDIRIZZO: ) (NOME: ) (TESTO:)
```

L’argomento **INDIRIZZO** specifica il nome del file. Il file deve trovarsi nella cartella “**estensioni**” (o una sua sottocartella) della cartella in cui è installato Atomic. **N.B: la cartella d’installazione è diversa dalla cartella delle risorse**, per trovarla in modo semplice fai click sull’icona di Atomic con il tasto destro e clicca “Apri percorso file”.

L’argomento **NOME** indica il nome originale della funzione all’interno della DLL.

L’argomento **TESTO** specifica in un’unica stringa di testo il nome che avrà la funzione importata in Atomic e i sui argomenti. I valori specificati negli argomenti saranno quelli di default. Per specificare una stringa come argomento bisogna utilizzare il simbolo dell’apostrofo invece che le doppie virgolette. In base ai valori inseriti il programma capisce automaticamente se l’argomento dovrà essere un numero o un testo.

Esempio:

```
INIZIA
importa funzione esterna → (INDIRIZZO: "test.dll") (NOME: "cubo")
                           (TESTO: "ottieni il cubo di → (VALORE:0)")

importa funzione esterna → (INDIRIZZO: "test.dll") (NOME: "somma")
                           (TESTO: "ottieni la somma tra → (VALORE 1:0) (VALORE 2:0)")

CICLO CONTINUO
numero = 22
valore = ottieni il cubo di → (VALORE: numero)
disegna testo → (TESTO: "Il cubo di <numero> è <valore>")

a = 10
b = 3
valore2 = ottieni la somma tra → (VALORE 1: a) (VALORE 2: b)
disegna testo → (TESTO: "la somma tra <a> e <b> equivale a <valore2>") (Y: 175)
```

Codice della DLL (C++):

```
#define Atomic_export extern "C" __declspec (dllexport)

Atomic_export double cubo(double n){
    return n*n*n;
}

Atomic_export double somma(double a, double b) {
    return a + b;
}
```

L’argomento NOME e i tipi di dato per gli argomenti devono per forza essere forniti dal programmatore che ha creato la DLL mentre il nome della DLL, il nome della funzione e le etichette degli argomenti possono essere cambiati a piacimento.

Esempio:

```
INIZIA
importa funzione esterna ➔ (INDIRIZZO: "MELONE.dll") (NOME: "cubo")
                           (TESTO: "ottieni BANANA ➔ (Z:0)")

importa funzione esterna ➔ (INDIRIZZO: "MELONE.dll") (NOME: "somma")
                           (TESTO: "ottieni MELA ➔ (X:0) (Y:0)")

CICLO CONTINUO
numero = 22
valore = ottieni BANANA ➔ (Z: numero)
disegna testo ➔ (TESTO: "La banana di <numero> è <valore>")

a = 10
b = 3
valore2 = ottieni MELA ➔ (X: a) (Y: b)
disegna testo ➔ (TESTO: "la mela tra <a> e <b> equivale a <valore2>") (Y: 175)
```

Presumibilmente in futuro saranno utilizzabili allo stesso modo dei file .dll i file .dylib su MacOS e i file .so su Linux.

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. Game Maker Studio è un progetto longevo, convalidato e apprezzato in tutto il mondo; inoltre è già utilizzato con successo anche a scopo didattico. L'unico svantaggio di questo approccio di sviluppo molto veloce e user friendly è che il codice scritto non è ottimizzato. 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 e docenti, interessati ad ampliare il progetto.

COME COLLABORARE AL PROGETTO COME SVILUPPATORE

Ci sono tre modi per collaborare attivamente al progetto come sviluppatore:

- scrivere tutorial e librerie in linguaggio Atomic

- Scrivere DLL in C++ per estendere le funzionalità di Atomic

- Scrivere nuove funzioni in GML o (in futuro) prendere parte allo sviluppo del core in GML o in C++.

Per maggiori informazioni scrivi a info@bergame.eu

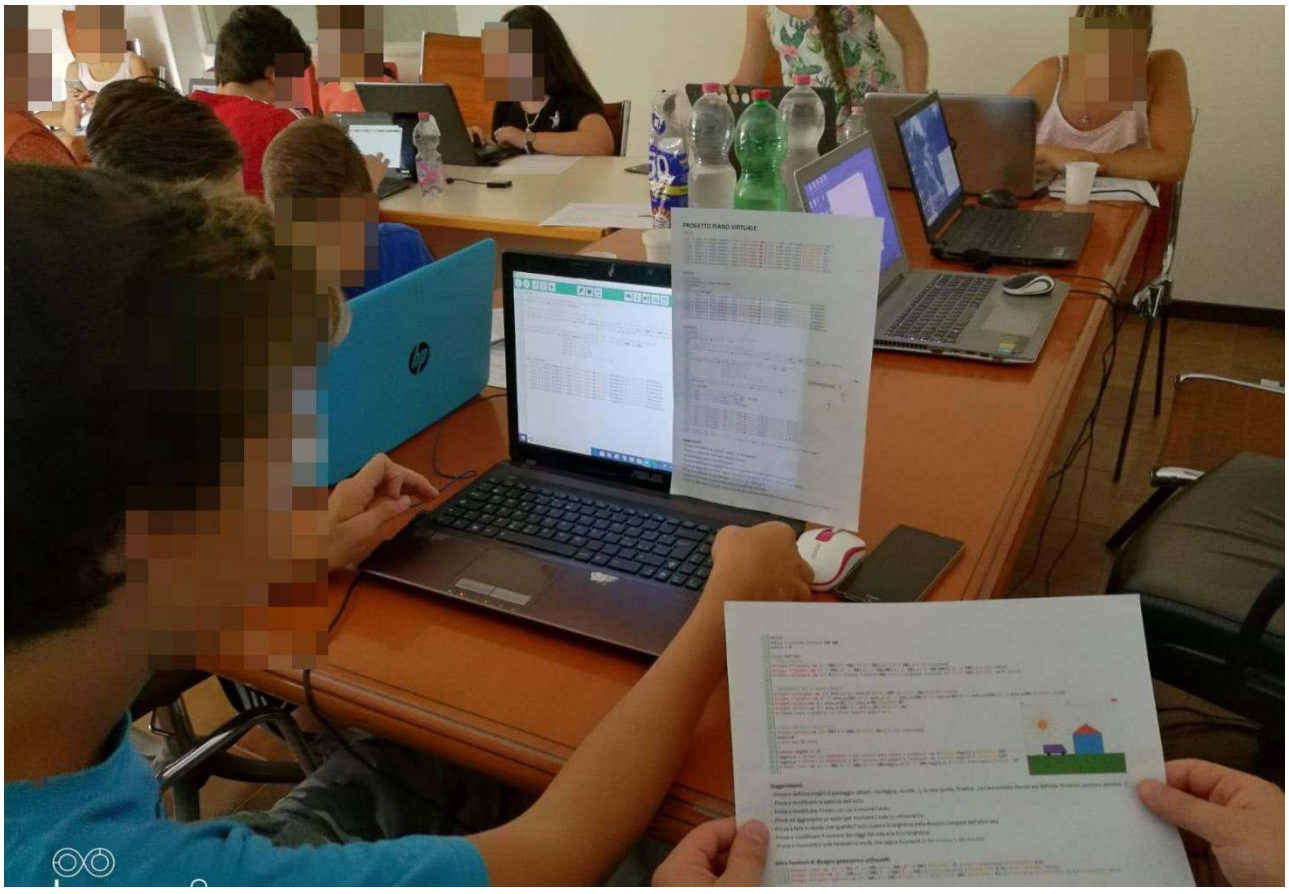
COME COLLABORARE AL PROGETTO COME DOCENTE

L'azione di collaborazione più vitale per i docenti è molto semplice: usare Atomic! Non importa se a scuola, in un contesto associativo o in un corso privato: qualsiasi utilizzo di Atomic è ben auspicato e incoraggiato.

Il sito ufficiale del progetto è www.atomicc.it, in questo sito i docenti possono avere un ruolo attivo commentando gli articoli e i tutorial, raccontando le loro esperienze e dando suggerimenti e consigli. Su richiesta i docenti possono anche pubblicare articoli, per esempio possono pubblicare i progetti dei propri alunni o raccontare le loro esperienze sul coding e sull'utilizzo di Atomic.

Un'altra iniziativa a cui un docente può prendere parte sono i **gruppi di coding**: dei gruppi informali che si ritrovano una volta al mese per un pomeriggio di coding; senza lezioni frontali ma soltanto con una traccia comune da seguire ed ampliare (metodo learning by doing).

Per creare un nuovo gruppo di coding Atomic nella tua città puoi scrivere a info@bergame.eu , riceverai tutte le informazioni su come trovare un posto per ospitare gli incontri e su come promuovere il gruppo.



ATOMIC IN SINTESI

ESPRESSIONI

Operatori aritmetici: +, -, *,/,^
È possibile utilizzare valori interi e decimali, le parentesi tonde, variabili e costanti.

Esempio:

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

DICHIARARE/MODIFICARE UNA VARIABILE GLOBALE

Sintassi:

x = y
nome variabile = valore

Esempi:

```
gatto = 5
cane = vero
topo = "ciao!"
```

COMMENTI

Sintassi:

//commento su singola linea
/*commento
multilinea*/

Esempi:

```
//gatto = 5  questo codice verrà ignorato
/*cane = vero
topo = ciao questo codice verrà ignorato*/
```

FUNZIONI

Sintassi:

f --> (X: a) (Y: b) …
nome funzione --> (NOME ARGOMENTO: valore) (NOME ARGOMENTO: valore)

Oppure tramite sintassi abbreviata:

f (X: a, Y: b …)
nome funzione (NOME ARGOMENTO: valore, NOME ARGOMENTO: valore)

Esempi:

```
disegna cerchio → (COLORE: rosso) (RAGGIO: 250)
disegna cerchio (COLORE: rosso, RAGGIO: 250)
```

DICHIARARE/MODIFICARE UNA VARIABILE GLOBALE TRAMITE UNA FUNZIONE CHE RESTITUISCE UN VALORE

Sintassi:

x = f --> (X: a) (Y: b) …
x = nome funzione --> (NOME ARGOMENTO: valore) (NOME ARGOMENTO: valore)

Oppure tramite sintassi abbreviata:

x = f (X: a, Y: b …)
x = nome funzione (NOME ARGOMENTO: valore, NOME ARGOMENTO: valore)

Esempi:

```
1  cane = ottieni la media tra ➔ (VALORE 1: 24) (VALORE 2: 15) (VALORE 3: 18)
2  cane = ottieni la media tra (VALORE 1: 24, VALORE 2: 15, VALORE 3: 18)
```

AUMENTARE/DIMINUIRE UNA VARIABILE

Sintassi:

aumenta x di y
diminuisci x di y

Esempio:

```
1  aumenta gatto di 5
2  diminuisci cane di 11
```

ISTRUZIONI CONDIZIONALI

se, finche, ripeti per x volte

Sintassi:

C {…}
Condizione { blocco di istruzioni }

Esempi:

```
1  se cane = 2 { disegna cerchio }
2  finche cane > 2 { disegna cerchio }
3  ripeti per 2 volte { disegna cerchio }
```

LEGGERE LE VARIABILI LOCALI DEGLI OGGETTI

Sintassi:

x del y (del/dell'/dello/della)
variabile dell'oggetto

Esempi:

```
1  bersaglio del giocatore
2  benzina dell'auto
3  ROTAZIONE della palla
```

CREARE E MODIFICARE OGGETTI/ESEMPLARI

Sintassi:

f --> (CARATTERISTICA: valore) (variabile locale: valore)

CARATTERISTICA = variabile già integrata nell'oggetto (NOME, OGGETTO, GENITORE, X, Y, Z, IMMAGINE, SCALA ASSE X, SCALA ASSE Y, TRASPARENZA, COLORE, VELOCITA, DIREZIONE, ROTAZIONE)

variabile locale = caratteristica extra dell'oggetto dal nome e valore arbitrario.

Esempi:

```
crea un oggetto ➔ (NOME: palla) (COLORE: verde) (peso: 20)
crea un esemplare ➔ (NOME: automobile) (VELOCITA: 0) (benzina: 10)
modifica un elemento ➔ (NOME: automobile) (VELOCITA: 0) (benzina: benzina-1)
```

DEFINIRE NUOVE FUNZIONI

Sintassi:

definisci funzione "nome funzione" { … }

Oppure con restituzione di un valore:

definisci funzione "ottieni … nome funzione"
{
…
con questa funzione ottieni …
}

Esempi:

```
definisci funzione "disegna casa"
{
disegna rettangolo ➔ (X: <"X">) (Y: <"Y">-<"ALTEZZA">) (BASE: <"LARGHEZZA">) (ALTEZZA: <"ALTEZZA">) (COLORE: <"COLORE">)
disegna triangolo ➔ (X 1: <"X">) (Y 1: <"Y">-<"ALTEZZA">) (X 2: <"X">+<"LARGHEZZA">) (Y 2: <"Y">-<"ALTEZZA">)
(X 3: <"X">+<"LARGHEZZA">/2) (Y 3: <"Y">-<"ALTEZZA">-100/[<"ALTEZZA">/75]) (COLORE: rosso)
}

disegna casa ➔ (LARGHEZZA: 250) (ALTEZZA: 100) (X: 500) (Y: 300) (COLORE: azzurro)
disegna casa ➔ (LARGHEZZA: 50) (ALTEZZA: 200) (X: 300) (Y: 300) (COLORE: verde)
```

```
definisci funzione "ottieni area rettangolo"
{
area = <"BASE">*<"ALTEZZA">
con questa funzione ottieni area
}

area_rettangolo_a = ottieni area rettangolo ➔ (BASE: 100) (ALTEZZA: 75)
area_rettangolo_b = ottieni area rettangolo ➔ (BASE: 20) (ALTEZZA: 35)
```

INCLUDERE CODICE

Sintassi:

includi "…"

Esempio

```
includi "codici/esempio.txt"
```