ATOMIC

Coding testuale per bambini e ragazzi

Documentazione tecnica

v 1.50 - 28/08/2020

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.

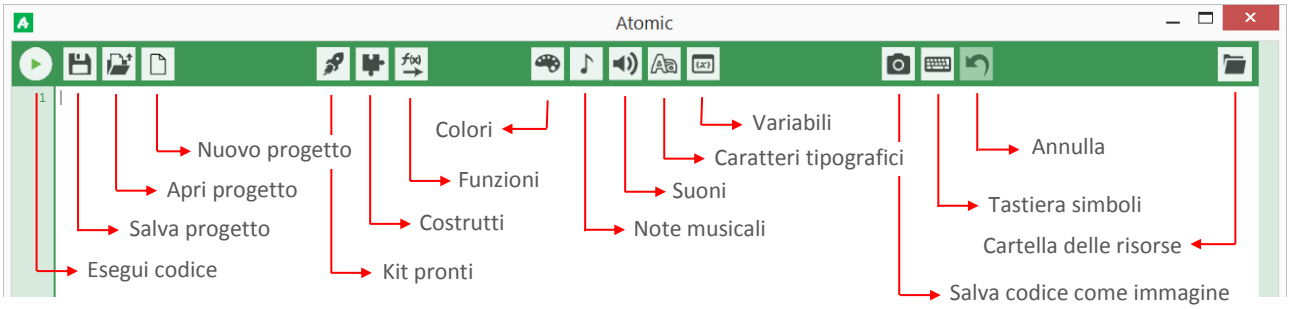
ATOMIC È UN LINGUGGIO PIÙ AVANZATO RISPETTO ALLA PROGRAMMAZIONE VISUALE?

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. Gli esperti concordano che una buona età per iniziare ad usare il pc per fare coding è 8-9 anni.

AMBIENTE DI SVILUPPO INTEGRATO

Avviando l'interprete di Atomic si apre un piccolo IDE (ambiente di sviluppo integrato):



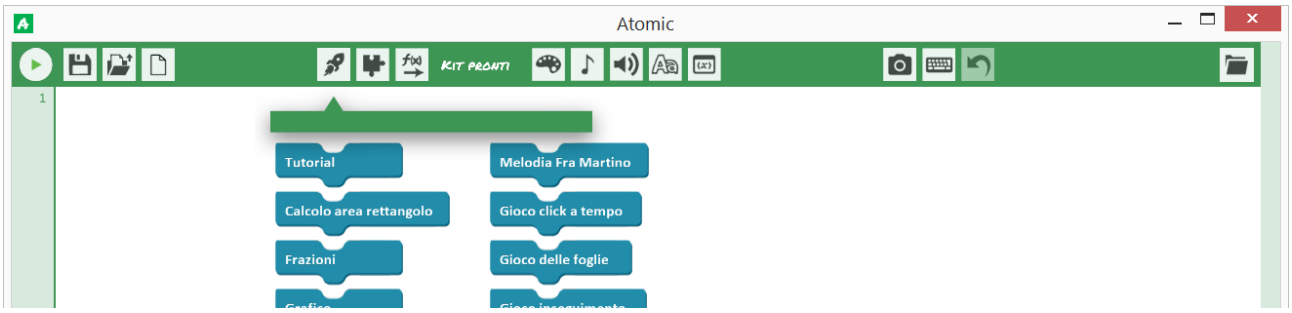
Le icone in alto (partendo da sinistra) sono:

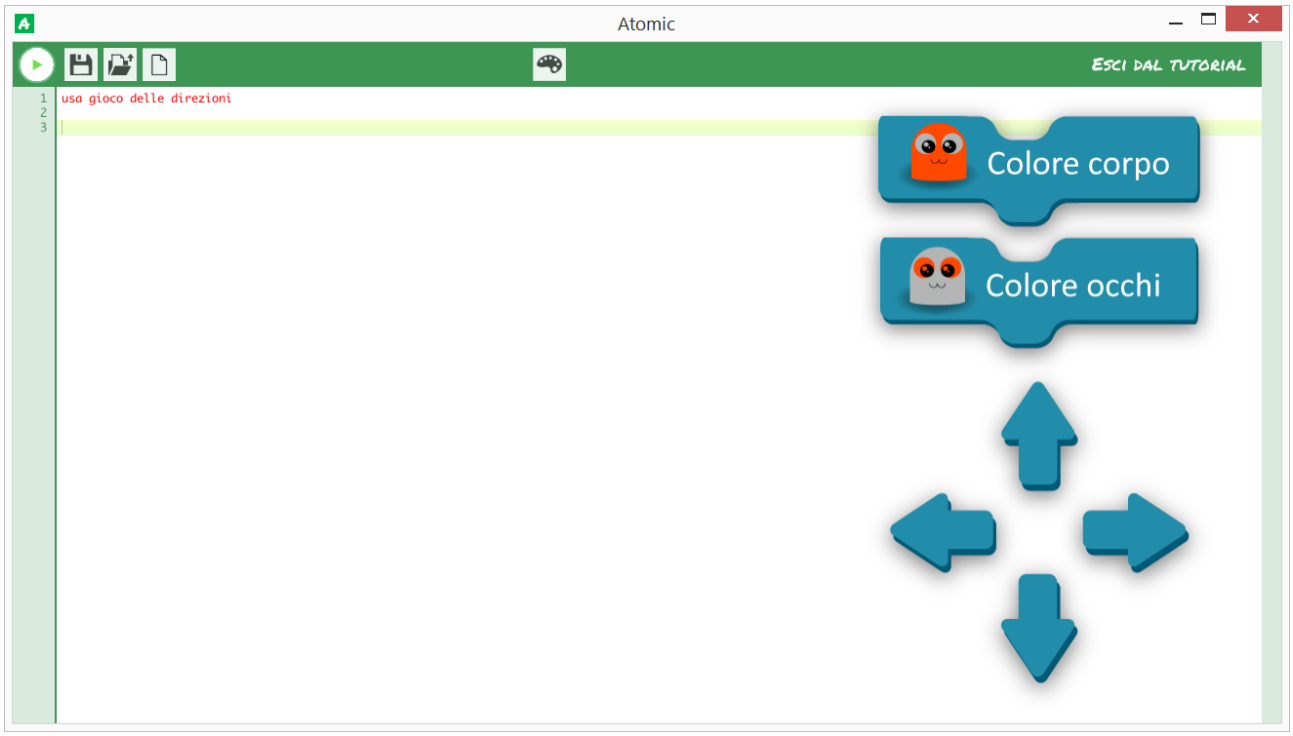
- **Esegui codice**: esegui il codice che hai scritto
- **Apri progetto**: apri un file di testo
- **Salva progetto**: salva il lavoro in un file di testo
- **Nuovo progetto**: apre un nuovo file di testo vuoto
- **Kit pronti:** apre un elenco di progetti d'esempio finiti; pronti da provare, modificare e personalizzare
- **Inserisci costrutti**: apre un elenco d'interfacce grafiche per inserire i costrutti principali del linguaggio
- **Inserisci funzioni**: apre un elenco di tutte le funzioni predefinite
- **Inserisci colori:** apre una tavolozza con tutti i colori predefiniti (definiti come costanti)
- **Inserisci note musicali:** apre una tastiera con le note musicali (definite come costanti)
- **Inserisci suoni**: apre un elenco di suoni predefiniti
- **Inserisci tipi di carattere**: apre un elenco di caratteri tipografici
- **Inserisci variabili**: apre un elenco di variabili predefinite
- **Esporta come immagine**: permette di salvare il codice come immagine (strumento per i docenti)
- **Tastiera simboli**: apre un interfaccia con vari simboli (inserisci simboli)
- **Annulla**: annulla l'ultima azione
- **Apri cartella risorse**: apre la cartella in cui inserire le risorse locali (immagini e suoni) per i propri progetti

Le icone **Costrutti**, **Funzioni**, **Colori**, **Note musicali, Suoni**, **Tipi di carattere**, **Variabili** , 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.

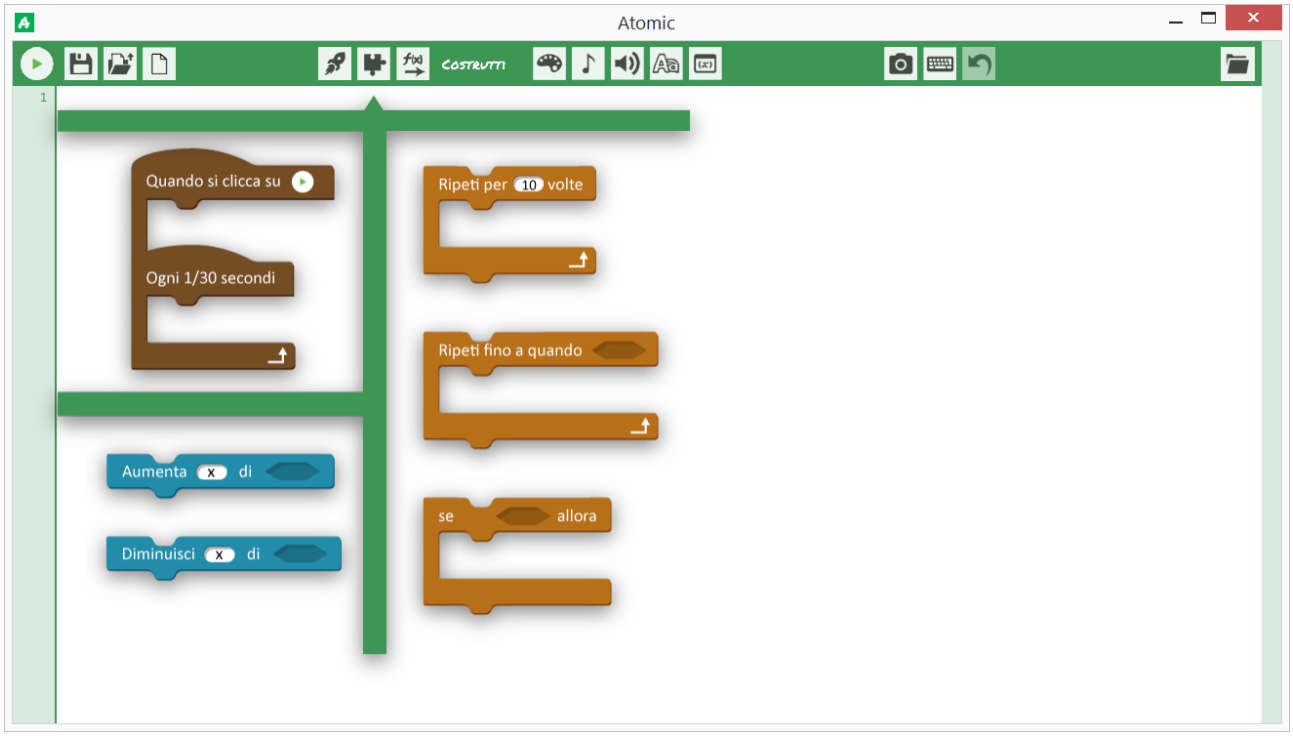
Le interfacce si rifanno alla **programmazione a blocchi**, simili a quelle utilizzate in **Scratch** e **Code.org**. Tuttavia, a differenza di quest'ultimi**, Atomic non utilizza la programmazione a blocchi ma converte istantaneamente il blocco selezionato in codice testuale e lo inserisce nella posizione 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 (come ad esempio nel Tutorial).

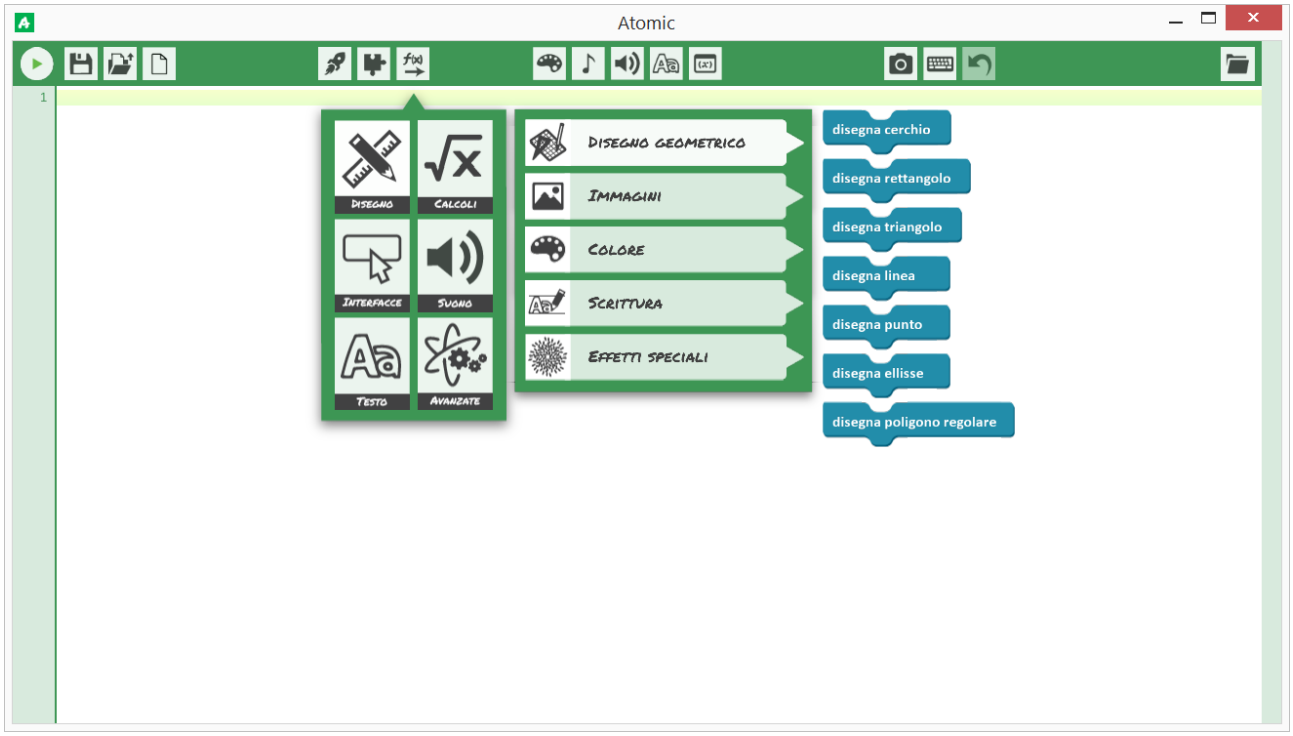




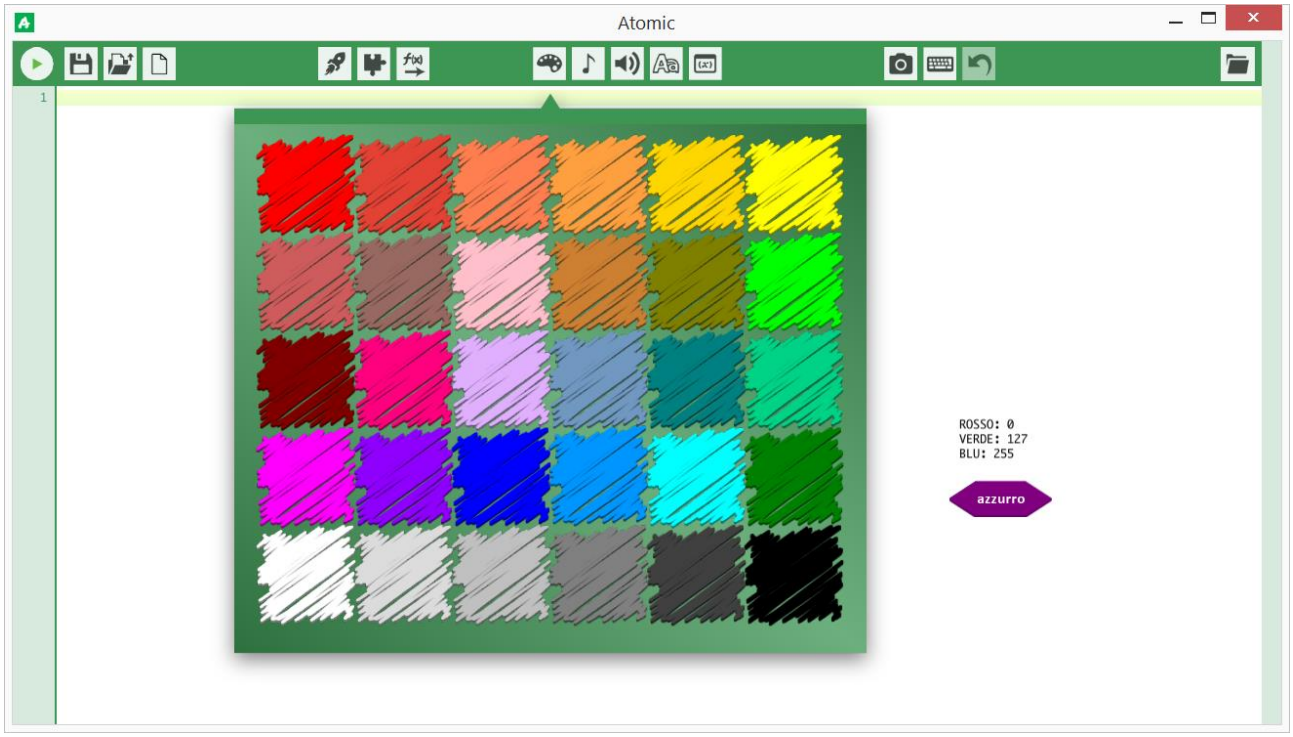
L'icona **Costrutti** permette d'inserire le strutture sintattiche di base di Atomic, gli "scheletri" su cui basare le proprie creazioni.



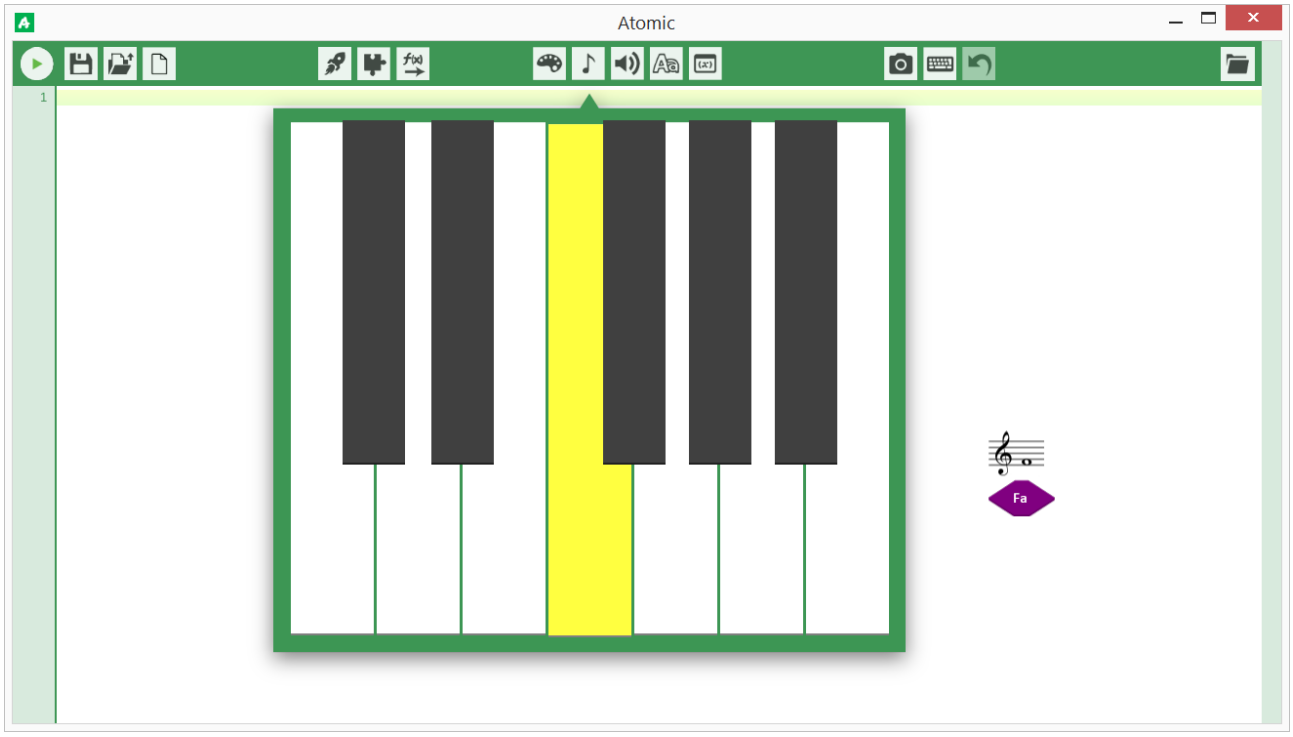
L'icona **Funzioni** permette di inserire tutte le funzioni predefinite di Atomic. Le funzioni sono divise per categoria, in modo da avere una buona panoramica delle "azioni" disponibili.



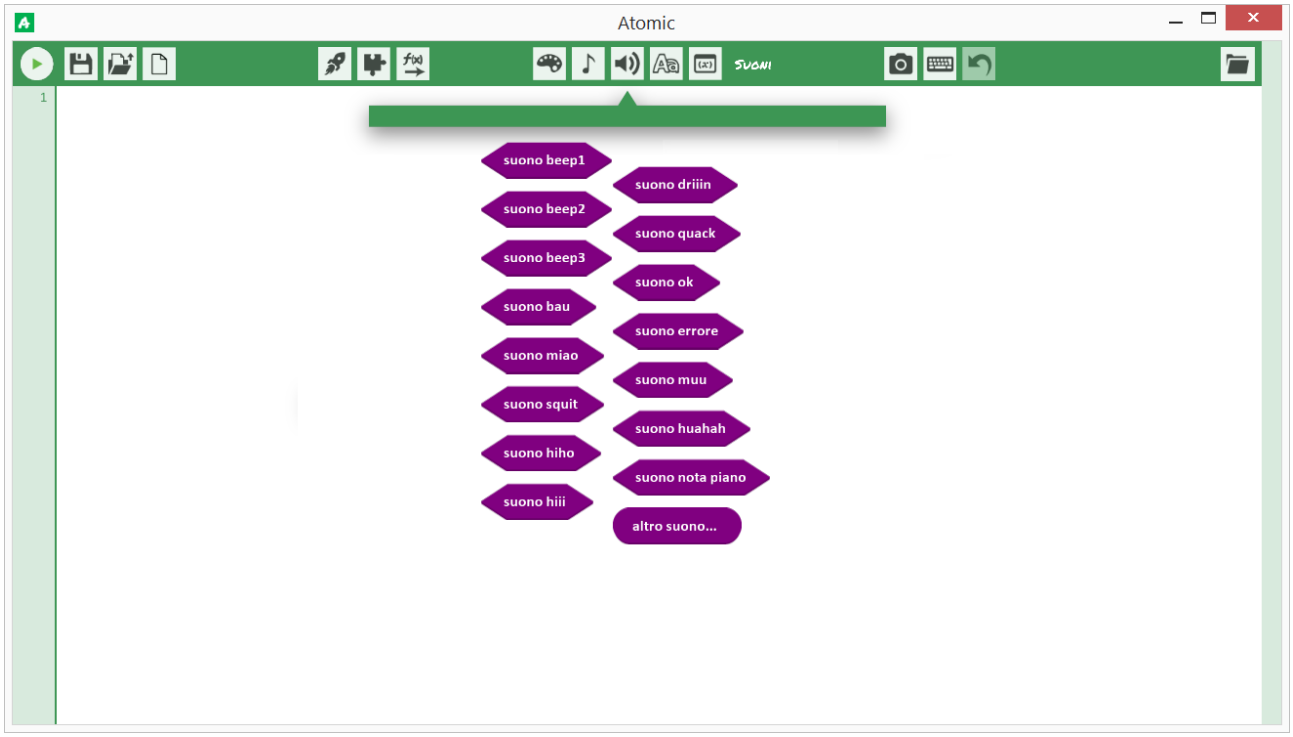
L'icona **Colori** permette di visualizzare ed inserire i colori di base disponibili ed esaminarne la composizione.



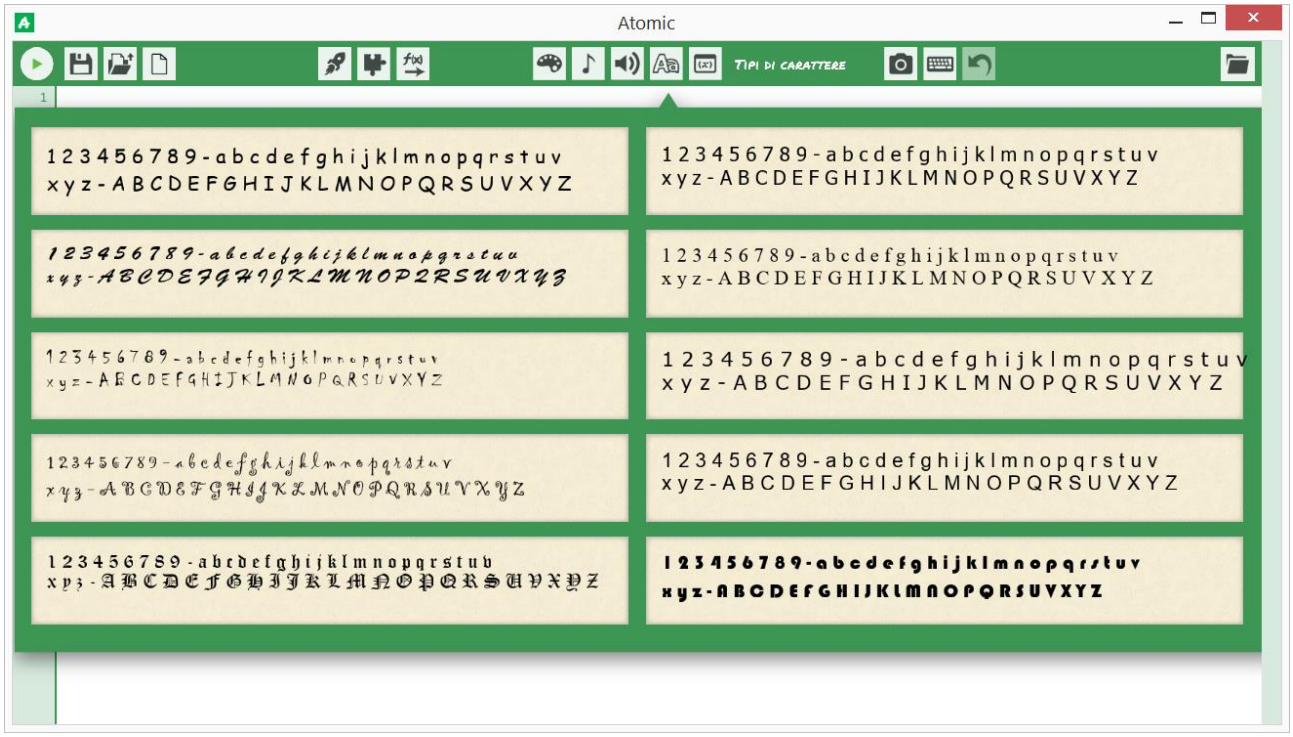
L'icona **Note musicali** permette di inserire e ascoltare l'anteprima delle note musicali.
Premendo il tasto **Ctrl** è possibile suonare la tastiera senza inserire le note.



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

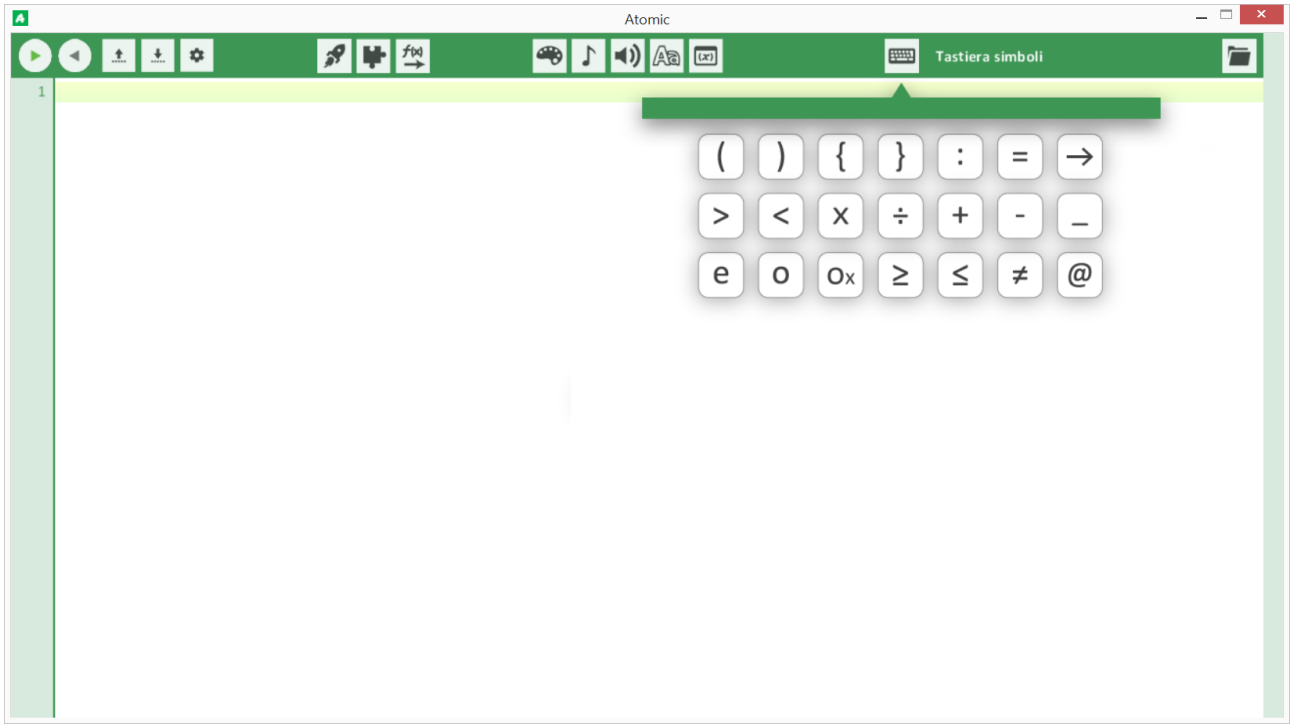


L'icona **Tipi di carattere** permette di visualizzare e inserire caratteri tipografici (font).

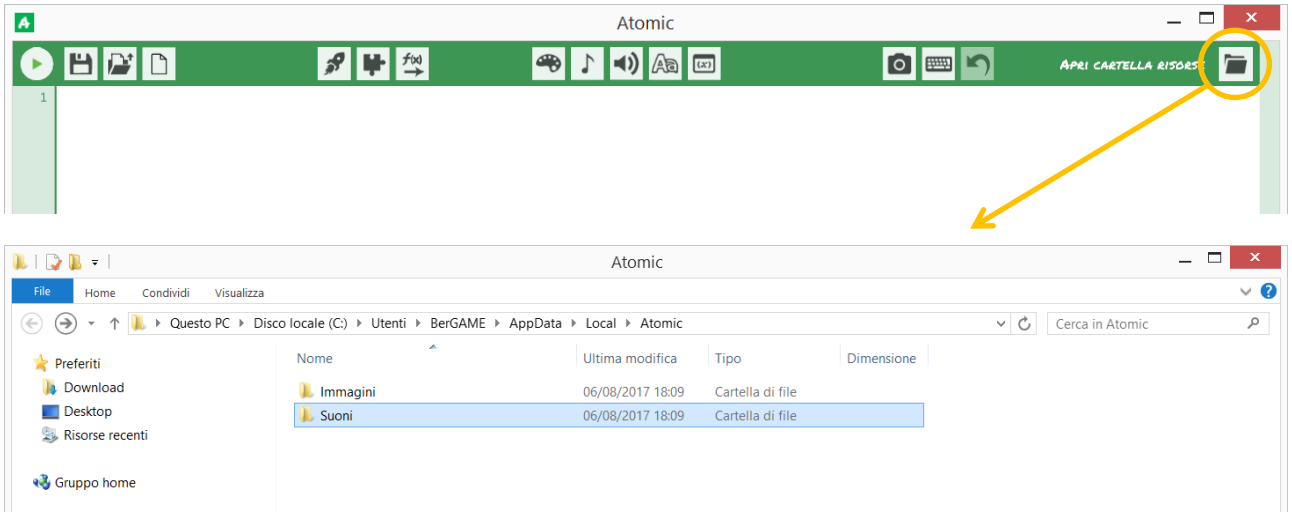


L'icona **Variabili** permette di visualizzare e inserire le variabili integrate.

L'icona **Tastiera simboli** permette d'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

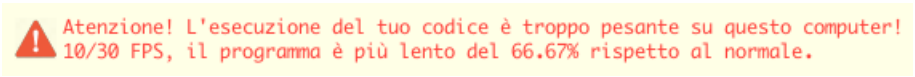
Atomic è un ambiente di sperimentazione sicuro. Con Atomic non è possibile creare programmi che danneggino il computer (sia involontariamente che volontariamente)

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 non adatto a scopi didattici 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

In informatica 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.
Atomic supporta senza errori qualsiasi numero dell'insieme $\mathbb{R}$ che abbia uno sviluppo decimale finito e un massimo di 16 cifre.

Esempio

```
1
2
3.56
257
1000
24525.23
```

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

I numeri razionali (insieme $\mathbb{Q}$) vengono trattati come espressioni.

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).

Esempio

```
"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 ...
}
```

O in alternativa la più comune sintassi semplificata:

```
INIZIA
... codice ...

CICLO CONTINUO
... codice ...
```

Esempio

```
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
```

Se gli eventi non vengono scritti tutto il codice viene eseguito automaticamente all'interno dell'evento CICLO CONTINUO.

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

ESPRESSIONI

In informatica un'espressione è un costrutto che utilizza numeri, variabili e costanti combinandoli con gli operatori aritmetici per restituire un risultato.

Gli operatori aritmetici utilizzabili sono i seguenti:

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

N.B. Bisogna tenere conto che i numeri reali utilizzati in informatica, per quanto affidabili, sono simulati: come avviene per le comuni calcolatrici, nei risultati che dovrebbero presentare uno sviluppo decimale infinito possono verificarsi degli arrotondamenti.

Esempio

```
3*2+2*5
```

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

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

Esempio

```
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 degli spazi tra gli elementi di una espressione; tuttavia non è possibile andare a capo. Ad esempio le seguenti forme di scrittura sono valide:

Esempio

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

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

Di seguito è riportato come esempio l'algoritmo per risolvere le equazioni di secondo grado; nel codice sono presenti diverse espressioni per eseguire i vari calcoli:

Esempio utilizzo estensivo di espressioni

```
//Formula per risolvere le equazioni di secondo grado: ax^2+bx+c=0
a = 1
b = -5
c = 6

//componiamo il testo del problema
disegna testo → (X: 25) (Y: 25)
                (TESTO: "Problema: risolvi l'equazione di secondo grado <a>x^2+<b>*x+<c>=0")

//Calcoliamo la discriminante (delta) dell'equazione. La formula è b^2-4ac
delta = b^2-(4*a*c)
radice_di_delta = ottieni la radice quadrata di → (VALORE: delta)
soluzione_discriminante = "Discriminante = <delta>"

//CASO 1: se la discriminante (delta) è maggiore di 0 allora calcola le due soluzioni
//Le forumule  sono:  (-b + radice quadrata di delta)/2a;  (-b - radice quadrata di delta)/2a;
se delta>0
{
s1=((b*-1) - radice_di_delta)/(2*a)
s2=((b*-1) + radice_di_delta)/(2*a)
soluzione="Soluzione 1: x=<s1> (a capo)Soluzione 2: x=<s2>"
}

//CASO 2: se la discriminante (delta) è uguale a 0 allora calcola una sola soluzione
//(Semplicemnte se delta equivale a 0 la sua radice sarà 0, quindi non ha senso sommarla o sottrarla)
se delta=0
{
s1=((b*-1))/(2*a)
soluzione="Soluzione: x=<s1>"
}

//CASO 3: se la discriminante (delta) è minore di 0 non ci sono soluzioni
se delta < 0 {soluzione="Soluzione: x= Numero non reale"}

disegna testo → (X: 25) (Y: 50) (TESTO: soluzione_discriminante)
disegna testo → (X: 25) (Y: 75) (TESTO: soluzione)
se delta < 0  {disegna testo → (X: 25) (Y: 125) (TESTO: "L'equazione non ha soluzioni reali!")}
```

VARIABILI

In informatica 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 di testo (una riga di testo). Esistono anche variabili integrate (predefinite) 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** (inizializzarla). Normalmente è buona prassi dichiararle nell'evento **INIZA**, tuttavia possono anche essere dichiarate nell'evento **CICLO CONTINUO**.

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:

Esempio

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

Tramite testo:

Esempio

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

Tramite costante:

Esempio

```
1 gatto = vero
2 gatto = Pi greco
3 colore = verde
```

Tramite altra variabile:

Esempio

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

Tramite espressione:

Esempio

```
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"):

Esempio

```
gatto = ottieni uno a caso di questi valori → (VALORE 1: 50) (VALORE 2: 100) (VALORE 3: 70)
gatto = ottieni la radice quadrata di → (VALORE: 324)
colore = ottieni uno a caso di questi valori → (VALORE 1: rosso) (VALORE 2: verde) (VALORE 3: blu)
```

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

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

Esempio

```
INIZIA
gatto = 50
timer 1 = 5

CICLO CONTINUO
se timer 1 > 0 allora disegna cerchio → (RAGGIO: gatto) .
se timer 1 = 0 allora gatto = "Ciao gatto!" disegna testo → (TESTO: gatto) .
```

E' possibile gestire gli spazi tra il simbolo "=" come si preferisce, le forme di scrittura riportate nell'esempio sono valide.

Esempio

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

Queste forme di scrittura "bizzarre" sono valide ma sconsigliate:

! Esempio

```
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

```
1 mio_colore = verde
```

Esempio

```
1 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 <figurine_iniziali> figurine.
Il giorno dopo va dal giornalaio e compra <pacchetti_comprati> pacchetti,
ogni pacchetto ne contiene <figurine_per_pacchetto>.
Nei <pacchetti_comprati> pacchetti che ha comprato ha
trovato <doppioni_trovati> doppioni che si tiene per sè.
Per completare l'album ci vogliono <figurine_completamento> 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 "
```

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:

! Esempio

```
problema = " Marco possiede <a> figurine.
Il giorno dopo va dal giornalaio e compra <b> pacchetti,
ogni pacchetto ne contiene <c>.
Nei <b> pacchetti che ha comprato ha trovato <d> doppioni che si tiene per sè.
Per completare l'album ci vogliono <f> 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 "
```

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. Per usarle non è necessario dichiararle con la sintassi *nome = valore*. Alcune possono essere modificate manualmente, altre vengono gestite automaticamente e possono essere solamente lette.

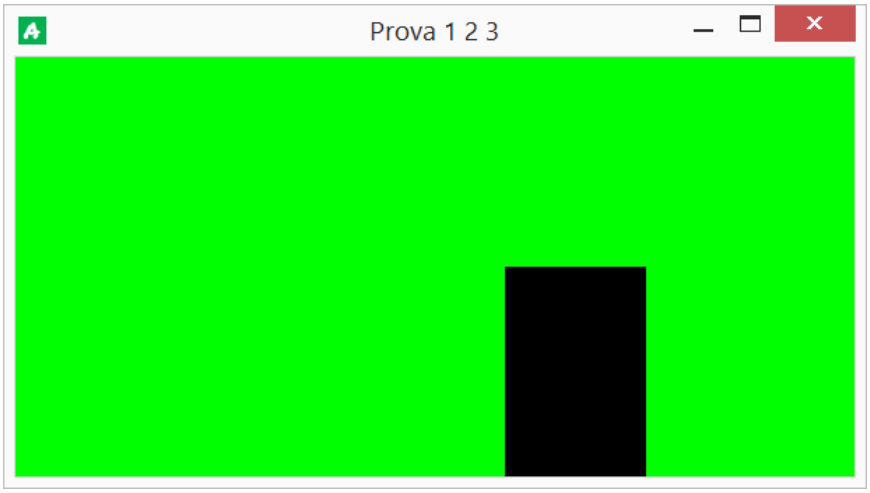
NOME VARIABILE	DESCRIZIONE	MODIFICABILE MANUALMENTE?
x del mouse	La posizione in pixel sull'asse x del cursore (freccetta) del mouse	no
y del mouse	La posizione in pixel sull'asse y del cursore (freccetta) del mouse	no
larghezza finestra	La larghezza in pixel della finestra	si
altezza finestra	L'altezza 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.

Per motivi di usabilità la variabile *larghezza finestra* gestisce correttamente la dimensione della finestra se ha un valore maggiore o uguale a 150; valori più piccoli non ridimensioneranno l'area e deformeranno la griglia.

Esempio

```
1  //Uso della variabile integrata "larghezza finestra"
2  larghezza finestra = 600
3
4  //Uso della variabile integrata "altezza finestra"
5  altezza finestra = 300
6
7  //Uso della variaibile integrata "colore sfondo"
8  colore sfondo = verde
9
10 //Uso della variabile integrata "nome finestra"
11 nome finestra = "Prova 1 2 3"
12
13 //Uso della variabile integrata "x del mouse"
14 disegna rettangolo → (X: x 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:

Esempio

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

che può anche essere scritta senza problemi in questo modo:

Esempio

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

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

Esempio

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

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

Esempio

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

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

Esempio

```
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).
Ad esempio la funzione *riavvia il programma* non richiede nessun argomento per funzionare.

Esempio

```
1 riavvia il programma
```

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 tra 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:)
```

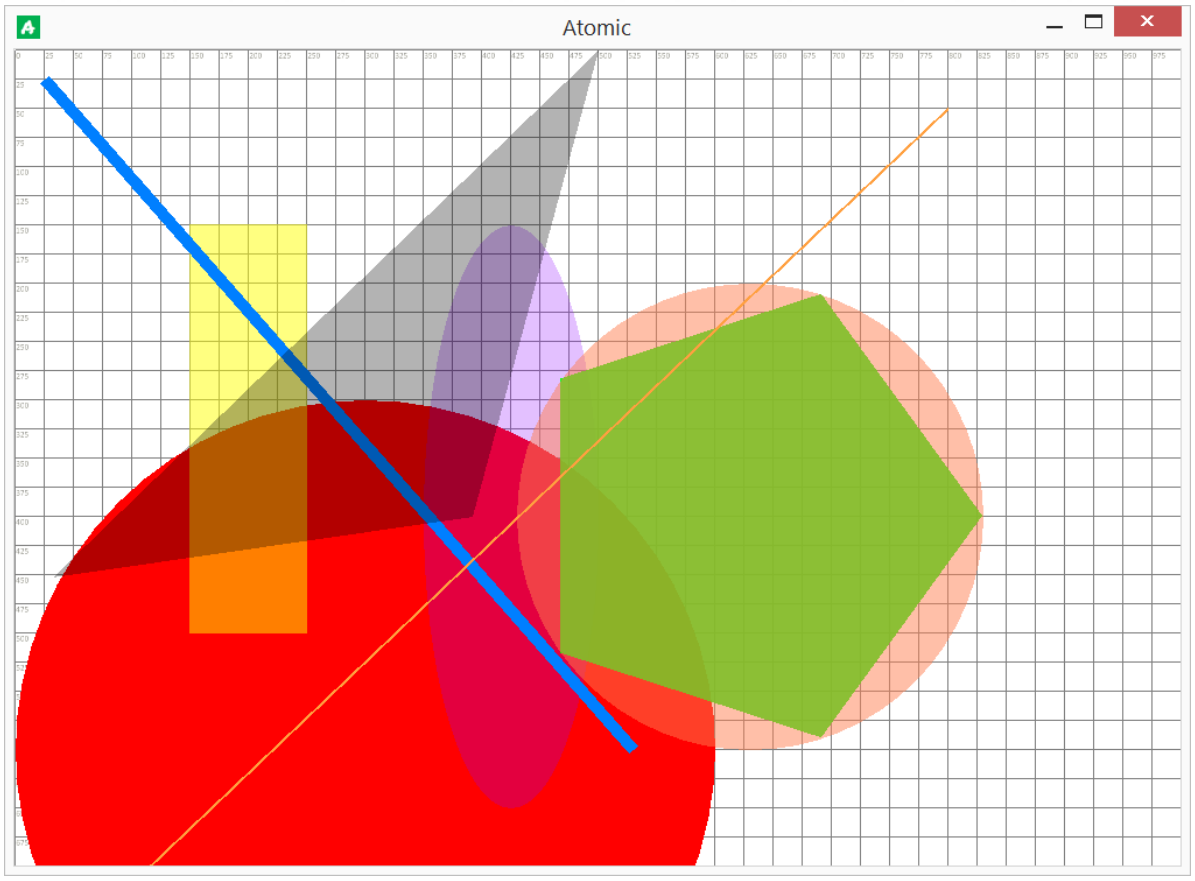
```
disegna poligono regolare → (X:) (Y:) (NUMERO LATI:) (RAGGIO:) (ROTAZIONE:) (COLORE:)
                            (TRASPARENZA:) (SOLO CONTORNO:)
```

```
disegna triangolo → (X 1:) (Y 1:) (X 2:) (Y 2:) (X 3:) (Y 3:) (COLORE:) (TRASPARENZA:)
                    (SOLO CONTORNO:)
```

Di seguito un disegno astratto realizzato con le funzioni geometriche di Atomic:

Esempio

```
1  disegna cerchio → (X: 300) (Y: 600) (RAGGIO: 300) (COLORE: rosso)
2  disegna rettangolo → (X 1: 200) (Y 1: 20) (BASE: 100) (ALTEZZA: 350)
3                       (RAGGIO: 300) (COLORE: giallo) (TRASPARENZA: 0.5)
4  disegna ellisse → (X 1: 350) (Y 2: 253) (X 2: 500) (Y 2: 650) (COLORE: viola) (TRASPARENZA: 0.25)
5  disegna linea → (X 1: 25) (Y 1: 25) (X 2: 530) (Y 2: 600) (COLORE: azzurro) (SPESSORE: 10)
6  disegna poligono regolare → (NUMERO LATI: 5) (RAGGIO: 300) (X: 630)
7                              (Y: 400) (COLORE: verde) (TRASPARENZA: 0.9)
8  disegna cerchio → (RAGGIO: 300) (X: 630) (Y: 400) (COLORE: corallo) (TRASPARENZA: 0.5)
9  disegna linea → (X 1: 800) (Y 1: 50) (X 2: 10) (Y 2: 800) (COLORE: arancione) (SPESSORE: 2)
10 disegna triangolo → (X 1: 500) (X 2: 32) (X 3: 392) (Y 1: 0) (Y 2: 452) (Y 3: 400)
11                     (COLORE: nero) (TRASPARENZA: 0.3)
12
```



Alcuni aspetti sul disegno che richiedono un chiarimento:

1) **l'origine della finestra** (il piano cartesiano) come in altri ambiti informatici riguardanti la grafica bidimensionale 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.

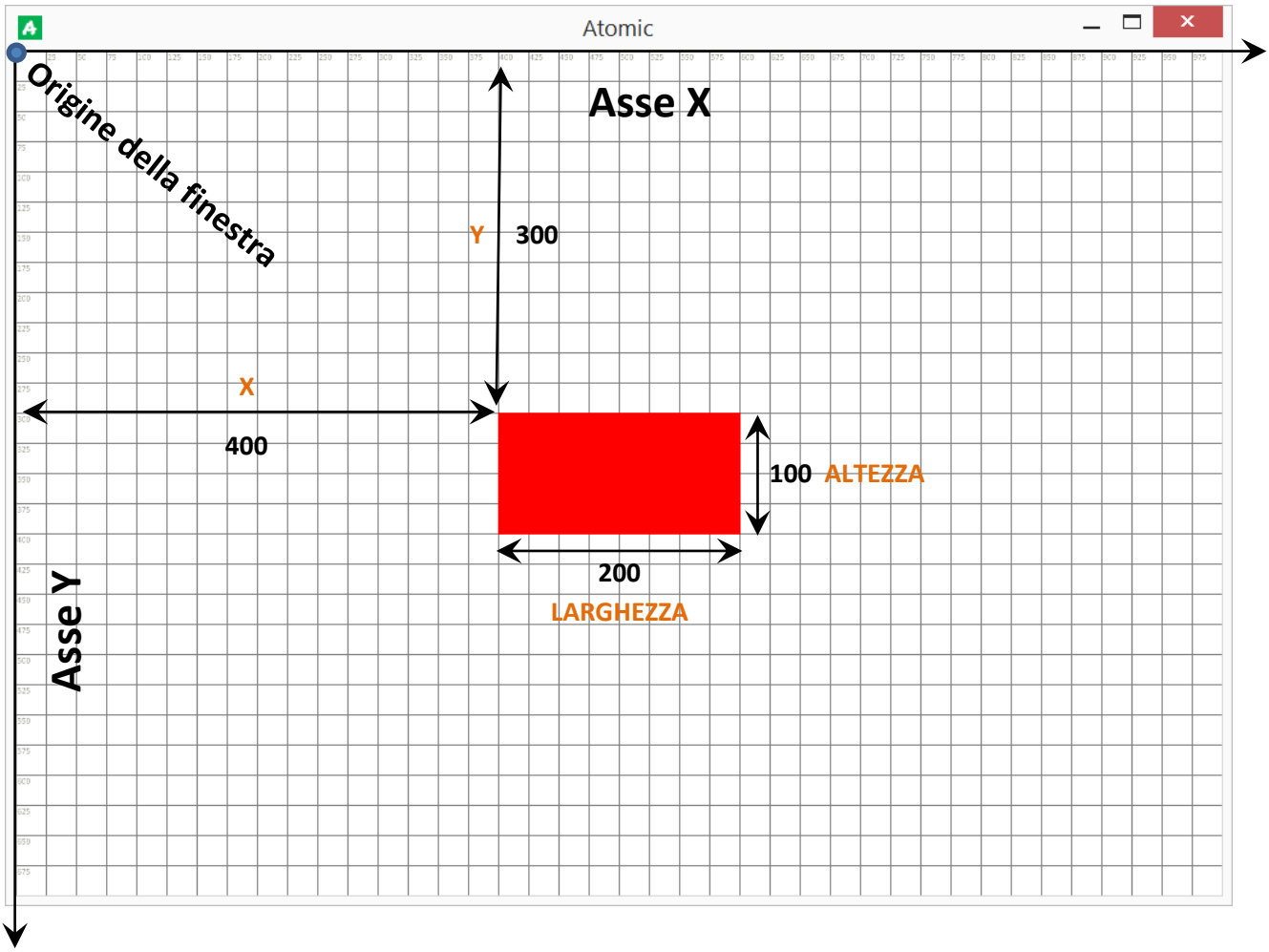
2) L'unità per esprimere le dimensioni in Atomic è una sola: il **pixel**. La griglia standard è composta da quadratini di 25 pixel.

3) L'argomento SOLO CONTORNO accetta solo due valori binari: **vero** o **falso**. Se si scrive (SOLO CONTORNO: vero) allora verrà disegnato solo il contorno della forma.

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 con il metodo RGB e HSV (vedi più avanti).

5) Se si utilizzano 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!


Atomic
Origine della finestra
Asse X
Y
300
X
400
100
ALTEZZA
200
LARGHEZZA
Asse Y

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 deve essere una **stringa** ovvero un testo racchiuso tra i simboli " " (virgolette, vedi sezione TIPI DI DATI). Se non si utilizzano questi simboli il programma cercherà di scrivere il valore di una variabile o una costante che abbia quel nome.

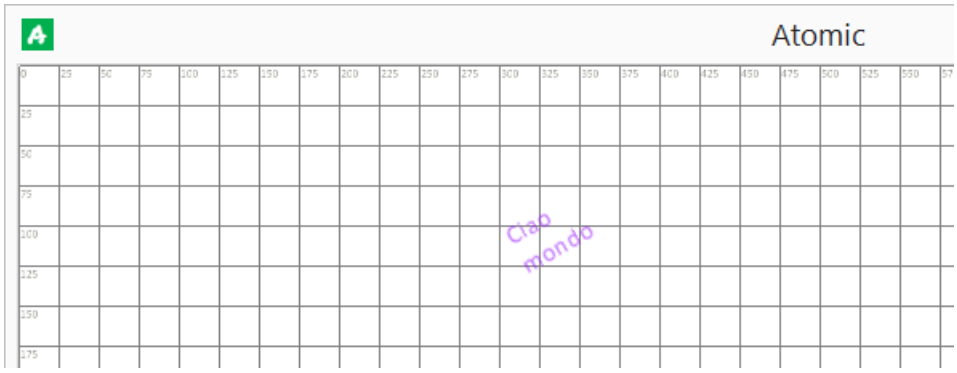
Gli altri argomenti sono interessanti per la realizzazione di "impaginazioni" elaborate, utili ad esempio all'introduzione del web design, mostrando come l'utilizzo degli editor testuali (in alternativa agli editor visuali) permetta il pieno controllo della grafica.

Esempio

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

Esempio

```
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)
```

Nell'esempio *stile_fumetto* 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.

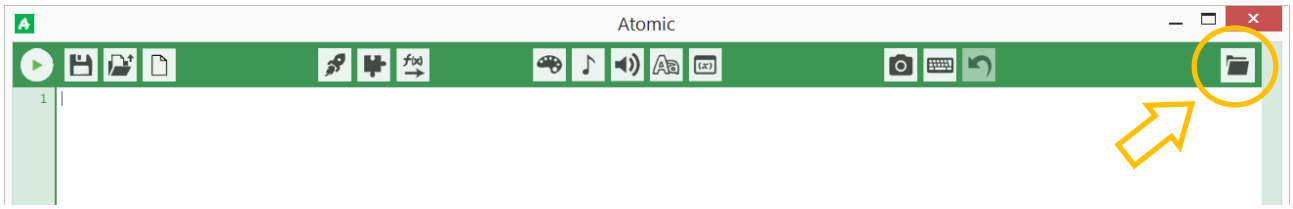
Funzioni di disegno delle immagini

Con la funzione ***disegna immagine*** è possibile disegnare qualsiasi immagine di formato .png e .jpg.
Ci sono vari modi per disegnare un'immagine che sono più o meno vantaggiosi in base al contesto.

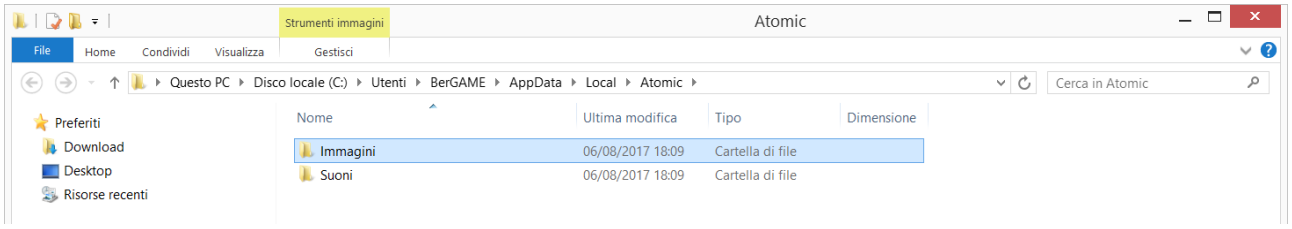
- Per gestire un'immagine ci sono due metodi: automatico o manuale.
- Le immagine possono essere delle risorse locali (salvate sul computer) o presenti su internet (online).

Risorse locali / metodo automatico

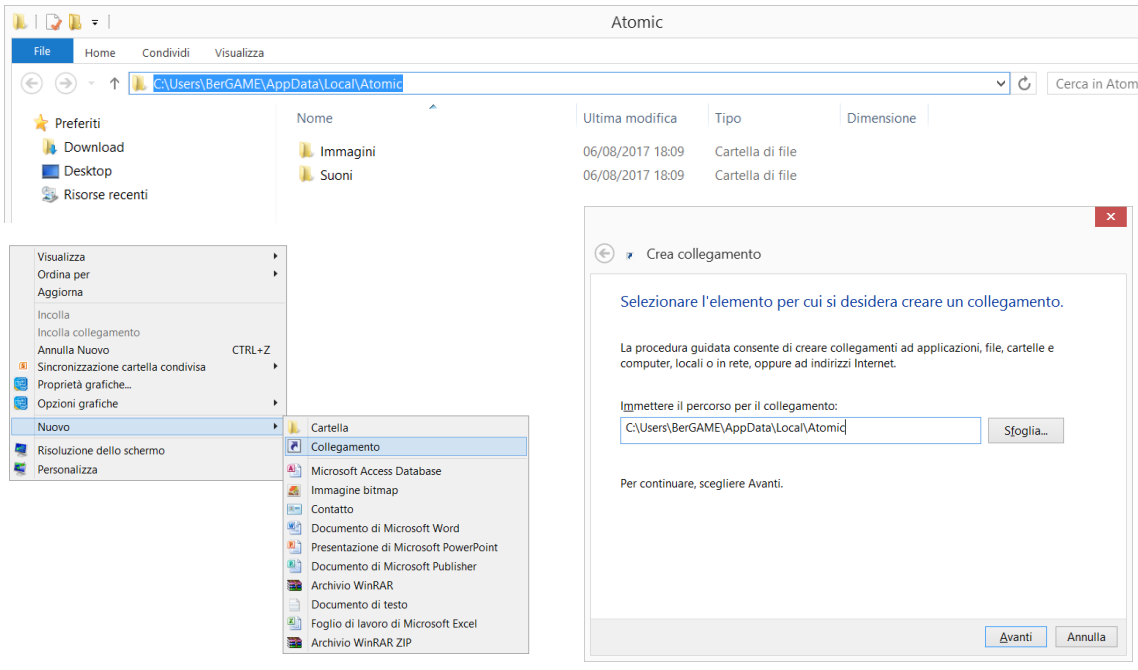
Le immagini per poter essere disegnate vanno inserite all'interno della **cartella risorse**. È possibile accedere a questa cartella dal mini editor cliccando sull'icona **Apri cartella risorse**.



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 se lo si desidera è possibile copiarlo e creare un comodo collegamento sul desktop (desktop -> click destro -> nuovo -> collegamento).



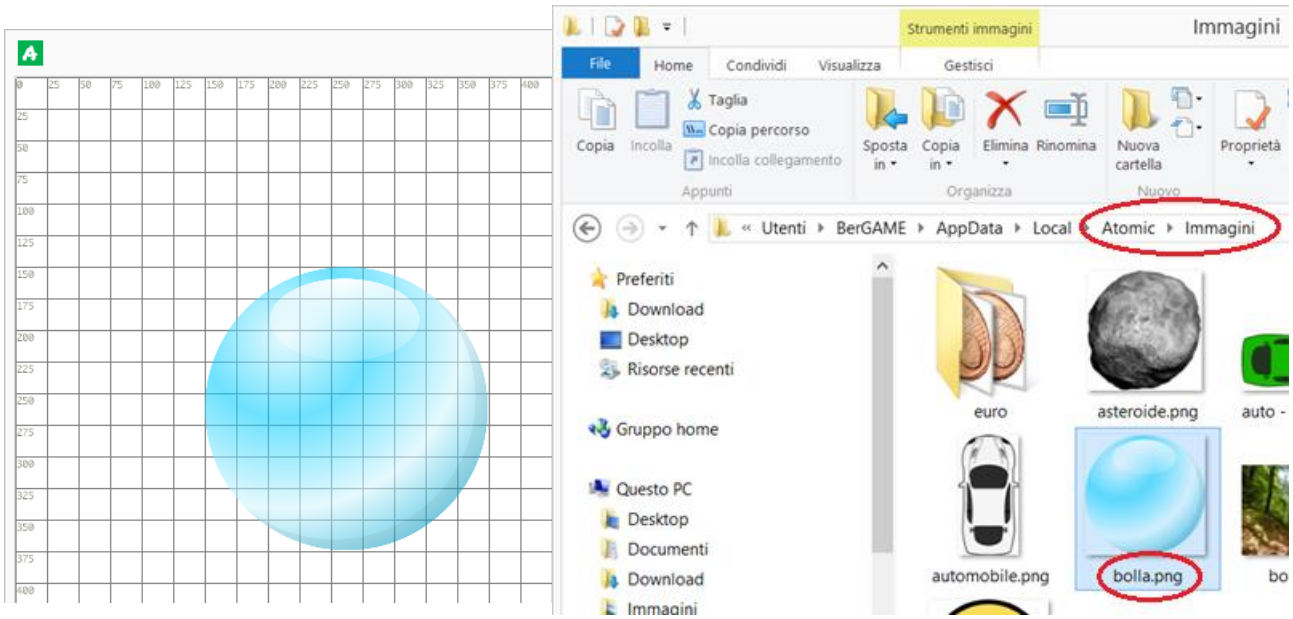
La funzione che disegna le immagini è ***disegna immagine***.

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

Una volta inserita un'immagine nella sottocartella "*immagini*" (nell'esempio "*bolla.png*" dentro "*Atomic/immagini*") Il modo più semplice per disegnarla è di utilizzare la funzione *disegna immagine* e l'indirizzo dell'immagine come valore dell'argomento **IMMAGINE**. L'indirizzo è **relativo**, ovvero parte dalla **cartella delle risorse** di Atomic.

Esempio

```
1  disegna immagine → (IMMAGINE: "immagini/bolla.png")
```



N.B. Di base i sistemi operativi Windows moderni nascondono l'estensione dei file (le ultime lettere dopo il nome che indicano il tipo di file, ad esempio .jpg, .png, .txt, …). Per evitare ambiguità è sempre meglio scrivere l'estensione del file ma se non si riesce a visualizzare l'estensione delle immagini non è un problema: **è possibile scrivere il nome del file senza l'estensione**; si occuperà Atomic di trovare l'estensione corretta.

Esempio

```
1  disegna immagine → (IMMAGINE: "immagini/bolla")
```

Risorse locali / metodo manuale

Disegnare direttamente un'immagine è molto comodo ma limitante: l'immagine viene gestita direttamente da Atomic e non è possibile modificarne il punto d'origine relativo.

Una volta inserita un'immagine nella cartella è possibile **memorizzarla all'interno di una variabile** utilizzando la funzione ***ottieni immagine***:

```
ottieni immagine → (NOME: ) (ORIGINE X:) (ORIGINE Y:) (FOTOGRAMMI:) (COLORE:) (TRASPARENZA:)
                   (SCALA ASSE X:) (SCALA ASSE Y:) (X:) (Y:) (ALTEZZA:) (LARGHEZZA:) (ROTAZIONE:)
```

Dopo aver inserito l'immagine *fotografia_bella.jpg* nella cartella immagini di Atomic è possibile scrivere:

Esempio

```
1 ottieni immagine → (NOME: "immagini/fotografia_bella")
```

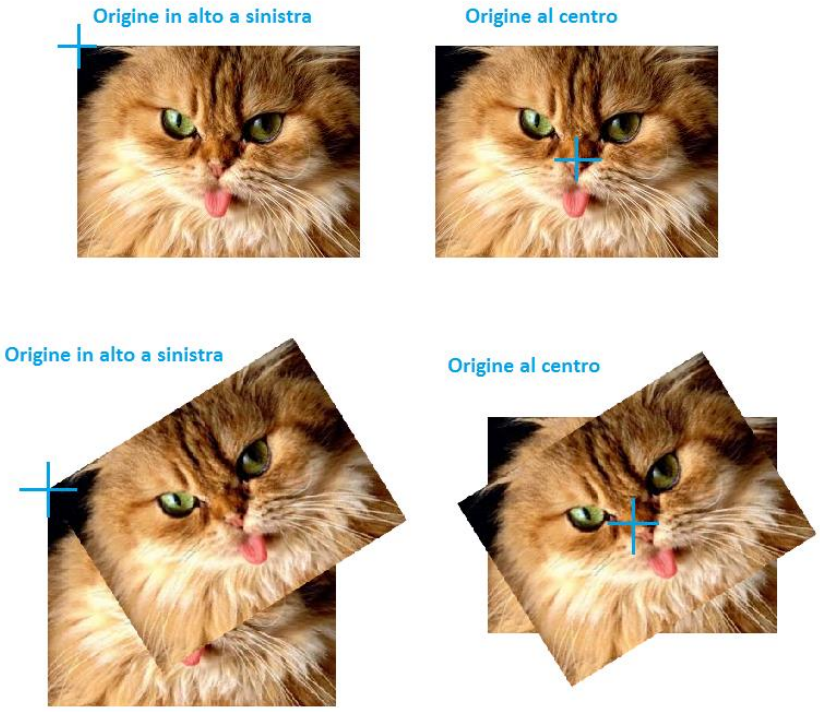
foto nell'esempio è la variabile al cui interno viene memorizzata l'immagine (è possibile usare qualsiasi nome valido come nome per la variabile). Il simbolo = che precede la funzione indica l'assegnazione di un dato alla variabile, in questo caso un dato complesso **ottenuto** da una funzione.

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 ottieni immagine → (NOME: "immagini/fotografia_bella") (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:

Esempio

```
1  ottieni immagine → (NOME: "immagini/immagini_gatti/gatto")
```

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

Esempio

```
1  foto = ottieni immagine → (NOME: "immagini/fotografia_bella") (ORIGINE X: 150) (ORIGINE Y: 100)
2  disegna immagine → (IMMAGINE: foto) (X: 75) (Y: 75)
```

O più correttamente:

Esempio

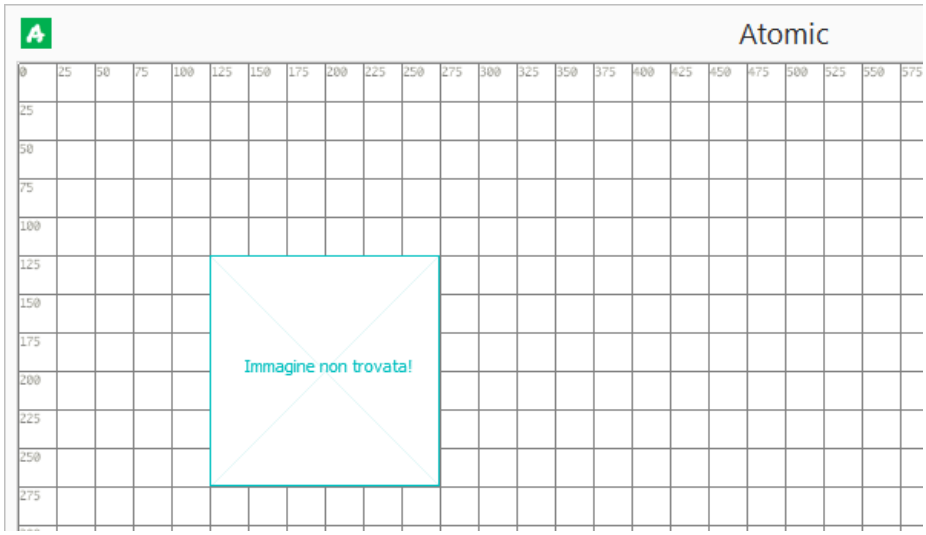
```
1  INIZIA
2  foto = ottieni immagine → (NOME: "immagini/fotografia_bella") (ORIGINE X: 150) (ORIGINE Y: 100)
3
4  CICLO CONTINUO
5  disegna immagine → (IMMAGINE: foto) (X: 75) (Y: 75)
```

Per gli argomenti (COLORE:) (TRASPARENZA:) (SCALA ASSE X:) (SCALA ASSE Y:) (X:) (Y:) (ALTEZZA:) (LARGHEZZA:) (ROTAZIONE:) della funzione "ottieni immagine" vedi la sezione "Modificare le immagini in fase di memorizzazione" poco più avanti.

Se un'immagine non esiste (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!".

! Esempio

```
1  foto = ottieni immagine → (NOME: "immagini/dasfsad")
2  disegna immagine → (IMMAGINE: foto) (X: 200) (Y: 200)
```



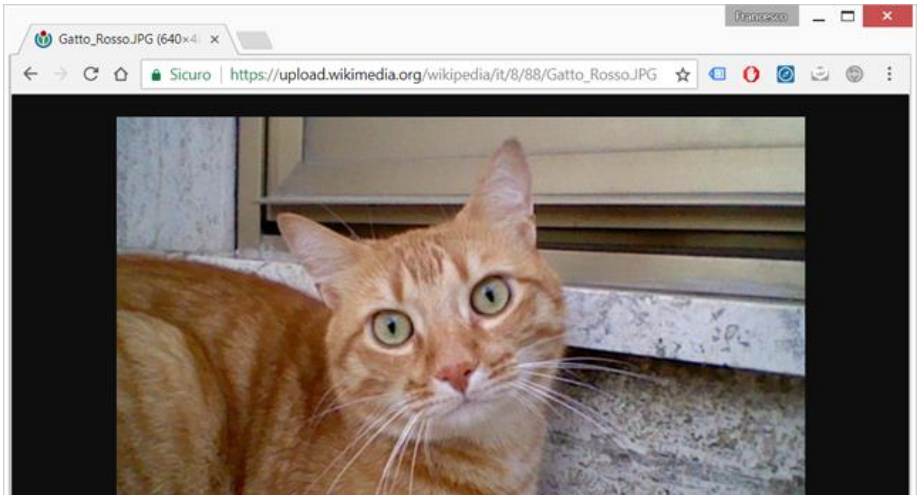
Atomic gestisce automaticamente le risorse duplicate: è impossibile caricare due volte la stessa risorsa; questo evita errori comuni che riguardano la gestione della memoria (RAM) che possono causare rallentamenti o crash del programma.

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:



I codici da utilizzare saranno:

Risorse online / metodo automatico

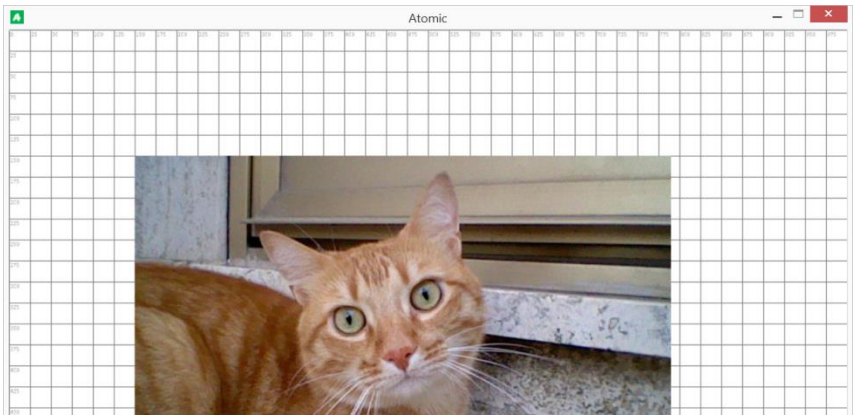
Esempio

```
1  disegna immagine → (IMMAGINE: "internet/upload.wikimedia.org/wikipedia/it/8/88/Gatto_Rosso.JPG")
```

Risorse online / metodo manuale

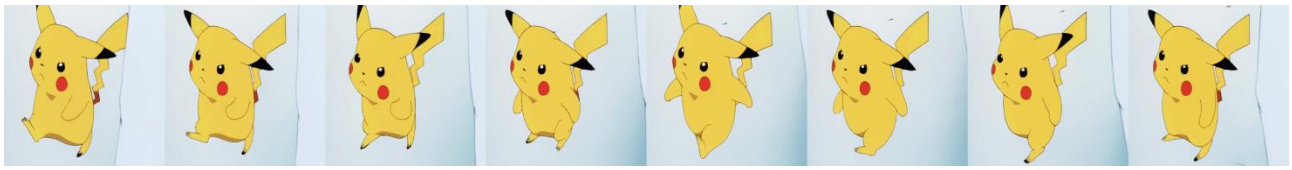
Esempio

```
1  foto = ottieni immagine → (NOME: "internet/upload.wikimedia.org/wikipedia/it/8/88/Gatto_Rosso.JPG")
2  disegna immagine → (IMMAGINE: foto) (X: 200) (Y: 200)
```



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

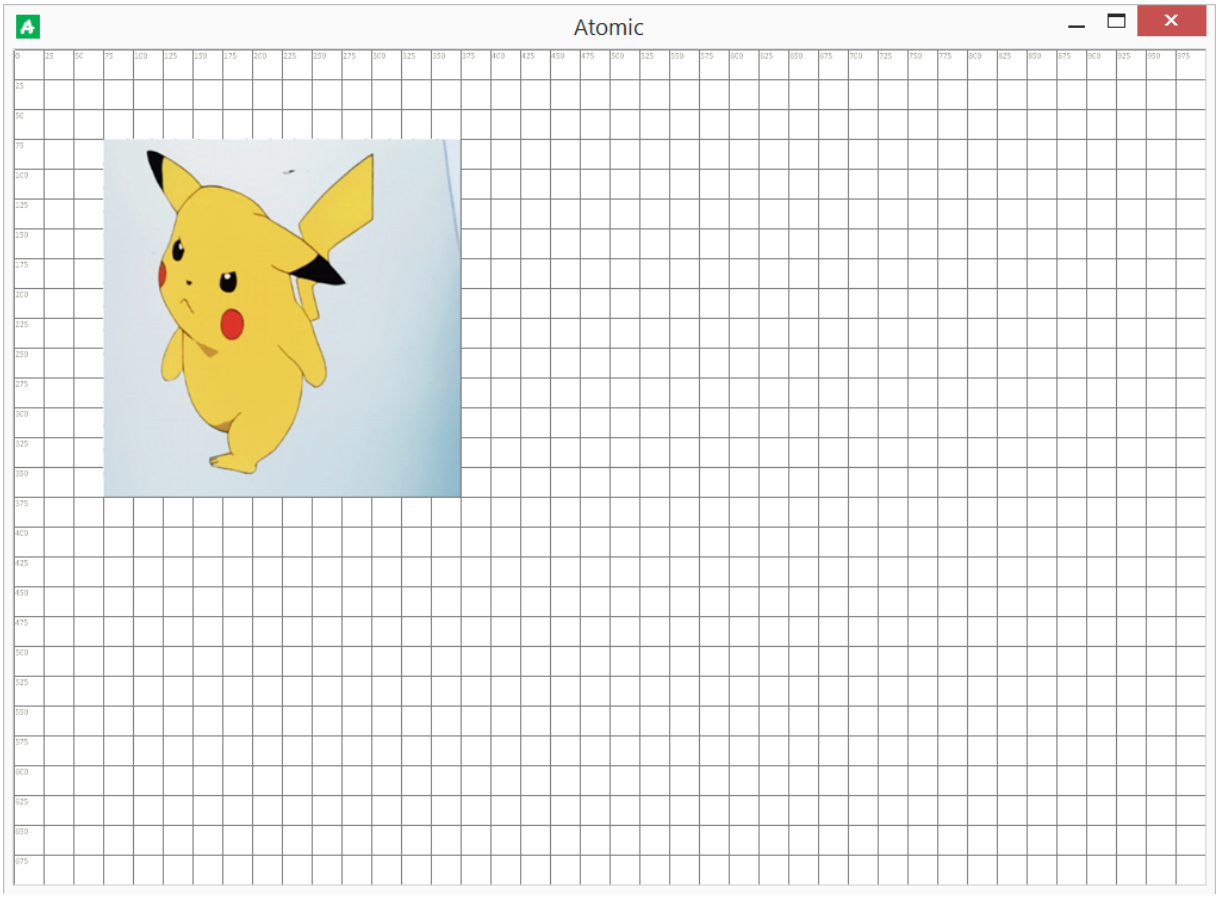
Una volta ottenuta la striscia basta specificare il **numero di fotogrammi** che contiene tramite l'argomento **FOTOGRAMMI** della funzione *ottieni immagine*.

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

```
1  foto = ottieni immagine → (NOME: "Immagini/pokemon.png") (FOTOGRAMMI: 8)
2  disegna immagine → (IMMAGINE: foto) (VELOCITA ANIMAZIONE: 15)
```



Disegnare fotogrammi di un'animazione

Per disegnare singoli fotogrammi di un'animazione esiste la funzione *disegna fotogramma*:

```
disegna fotogramma → (IMMAGINE:) (FOTOGRAMMA:) (X:) (Y:) (SCALA ASSE X:) (SCALA ASSE Y:)
                     (COLORE:) (TRASPARENZA:) (ROTAZIONE:)
```

usando questa funzione è possibile ottenere un'animazione di cui si ha il pieno controllo: ad esempio usando una variabile nell'argomento FOTOGRAMMA è possibile modificare la velocità di scorrimento tra singoli fotogrammi, invertire l'animazione, saltare dei fotogrammi, ripetere parti e creare nuove animazioni combinando fotogrammi non contigui.

Esempio

```
1  foto = ottieni immagine → (NOME: "Immagini/pokemon.png") (FOTOGRAMMI: 8)
2  disegna fotogramma → (IMMAGINE: foto) (FOTOGRAMMA: 5)
```

Modificare immagini in fase di memorizzazione

Per gli utenti più avanzati, tramite la funzione ***ottieni immagine*** è possibile modificare l'immagine prima della sua memorizzazione. È importante notare che questa pratica non modifica l'immagine originale memorizzata nel computer: le modifiche vengono applicate solo alla copia memorizzata temporaneamente nella RAM durante l'esecuzione del codice. Questa pratica è applicabile solo alle risorse grafiche locali (a causa del tempo di latenza non è possibile modificare un'immagine in streaming prelevata dal web).

```
ottieni immagine → (NOME: ) (ORIGINE X:) (ORIGINE Y:) (FOTOGRAMMI:) (COLORE:) (TRASPARENZA:)
                   (SCALA ASSE X:) (SCALA ASSE Y:) (X:) (Y:) (ALTEZZA:) (LARGHEZZA:) (ROTAZIONE:)
```

Tramite l'argomento COLORE è possibile colorare l'immagine.

Tramite l'argomento TRASPARENZA è possibile applicare un effetto di trasparenza all'immagine (valore compreso tra 0 e 1).

Tramite gli argomenti SCALA ASSE X e SCALA ASSE Y è possibile modificare le dimensioni e le proporzioni dell'immagine.

Tramite gli argomenti ALTEZZA e LARGHEZZA si definiscono le dimensioni dell'area di disegno in cui verrà rappresentata l'immagine, se questi argomenti non vengono specificati la dimensione dell'area di disegno corrisponderà a quella dell'immagine originale.

Tramite gli argomenti X e Y è possibile modificare il punto di origine assoluto (angolo in alto a sinistra) in cui verrà disegnata l'immagine all'interno dell'area di disegno.

Tramite l'argomento ROTAZIONE è possibile specificare la rotazione dell'immagine in gradi (valore compreso tra 0 e 360).

Le parti dell'immagine che non rientrano nell'area di disegno non verranno memorizzate.

Funzioni di casualità

Queste funzioni permettono di assegnare valori casuali alle variabili.

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

Esempio - Cerchio che vibra

```
x = ottieni uno a caso di questi valori → (VALORE 1: 100) (VALORE 2: 110) (VALORE 3: 120)
disegna cerchio → (X: x)
```

Esempio - Sfondo sfavillante

```
colore = ottieni uno a caso di questi valori → (VALORE 1: giallo) (VALORE 2: rosso)
colore sfondo = colore
```

```
ottieni un valore compreso tra questi → (VALORE 1:) (VALORE 2:)
```

```
ottieni un valore intero compreso tra questi → (VALORE 1:) (VALORE 2:)
```

Esempio - Effetto raffica casuale

```
larghezza1 = ottieni un valore compreso tra questi → (VALORE 1: 0) (VALORE 2: larghezza finestra)
larghezza2 = ottieni un valore compreso tra questi → (VALORE 1: 0) (VALORE 2: larghezza finestra)
larghezza3 = ottieni un valore compreso tra questi → (VALORE 1: 0) (VALORE 2: larghezza finestra)
disegna rettangolo → (X: 0) (Y: 100) (BASE: larghezza1)
disegna rettangolo → (X: 0) (Y: 200) (BASE: larghezza2)
disegna rettangolo → (X: 0) (Y: 300) (BASE: larghezza3)
```

Esempio - Cerchi colorati casuali

```
ripeti per 10 volte
{
x = ottieni un valore compreso tra questi → (VALORE 1: 0) (VALORE 2: larghezza finestra)
y = ottieni un valore compreso tra questi → (VALORE 1: 0) (VALORE 2: larghezza finestra)
colore = ottieni colore hsv → (TINTA: x)
disegna cerchio → (X: x) (Y: y) (COLORE: colore)
}
```

```
attiva casualità diversa per ogni avvio
```

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 per la prima volta (sempre lo stesso, con nessuna modifica) l'ordine della generazione casuale sarà sempre lo stesso. L'ordine viene poi modificato ad ogni nuova esecuzione ma sarà sempre lo stesso per quella determinata n esecuzione: la prima, la seconda, la terza, ecc, avranno sequenze casuali diverse tra loro ma saranno sempre le stesse. Questa sequenza si azzera chiudendo Atomic. Il motivo per cui la casualità di base non è totalmente casuale è per facilitare l'attività di debug.

Esempio

```
1  INIZIA
2  test = ottieni un valore intero compreso tra questi → (VALORE 1: 10) (VALORE 2: 5)
3
4  CICLO CONTINUO
5  disegna testo → (TESTO: test)
```

Supponiamo che la variabile *test* assuma il valore 8.
Ogni volta che il programma verrà avviato la prima volta il risultato sarà sempre 8.

Invece scrivendo:

Esempio

```
1  INIZIA
2  attiva casualità diversa per ogni avvio
3  test = ottieni un valore intero compreso tra questi → (VALORE 1: 10) (VALORE 2: 5)
4
5  CICLO CONTINUO
6  disegna testo → (TESTO: test)
```

Verosimilmente può succedere questo: la prima volta che il programma verrà avviato la variabile *test* assumerà il valore 8, la seconda volta (chiudendo e riaprendo Atomic) 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:)
```

```
ottieni il quoziente della divisione tra → (DIVIDENDO: ) (DIVISORE:)
```

divisione euclidea, si ottiene un numero intero

```
ottieni il resto della divisione tra → (DIVIDENDO: ) (DIVISORE:)
```

funzione *mod* (modulo), restituisce il resto di una divisione euclidea

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:)
```

```
ottieni potenza modulare di --> (VALORE: ) (ESPONENTE: ) (DIVISORE: )
```

Funzioni sul testo (stringhe di testo)

Il testo, inteso 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

```
1  prova = ottieni il simbolo di un testo a una determinata posizione → (TESTO: "Ciao Atomic")
2                                                                       (POSIZIONE: 4)
3  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

```
1  frase = "ciao ciao Atomic... ancora ciao!"
2  prova = ottieni il numero di parole specificate in un testo → (TESTO: frase) (PAROLA: "ciao")
3  prova2 = ottieni il numero di parole specificate in un testo → (TESTO: frase) (PAROLA: "a")
4  disegna testo → (TESTO: prova) (Y: 100)
5  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  lunghezza = ottieni lunghezza testo → (TESTO: "testo di prova")
2  disegna testo → (TESTO: lunghezza) (Y: 200)
```

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

```
testo = ottieni testo con parola sostituita →
(TESTO: "I gatti sono animali. I gatti sono mammiferi. I gatti sono belli.")
(PAROLA: "gatti") (NUOVA PAROLA: "cani")

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

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

```
testo = ottieni testo con parola sostituita solo la prima volta →
(TESTO: "I gatti sono animali. I gatti sono mammiferi. I gatti sono belli.")
(PAROLA: "gatti") (NUOVA PAROLA: "cani")

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

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

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

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

```
1  testo = ottieni un pezzo di questo testo → (TESTO: "messaggio di prova") (POSIZIONE: 4) (LUNGHEZZA: 6)
2  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

```
1  testo = ottieni posizione parola → (TESTO: "messaggio di prova") (PAROLA: "prova")
2  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 testo = ottieni testo maiuscolo → (TESTO: "messaggio di prova")
2 disegna testo → (TESTO: testo)
```

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

Inserire espressioni in un 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

```
età = 10
disegna testo → (TESTO: "Ciao, io ho <età> anni")
```

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

Esempio

```
disegna testo → (TESTO: "Sei per sei fa <6*6>")
```

Questo codice disegnerà il testo *"Sei per sei fa 36"*

Esempio

```
a = 10
b = 5
disegna testo → (TESTO: "<a+b*20-3>")
```

Questo codice disegnerà come testo il risultato dell'espressione a+b*20-3, ovvero *"107"*

Testo a capo in un testo

Se si vuole ottenere del testo a capo in una stringa di testo (senza mandare a capo il codice nell'editor) è possibile scrivere al suo interno *(a capo)*.

Esempio

```
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.

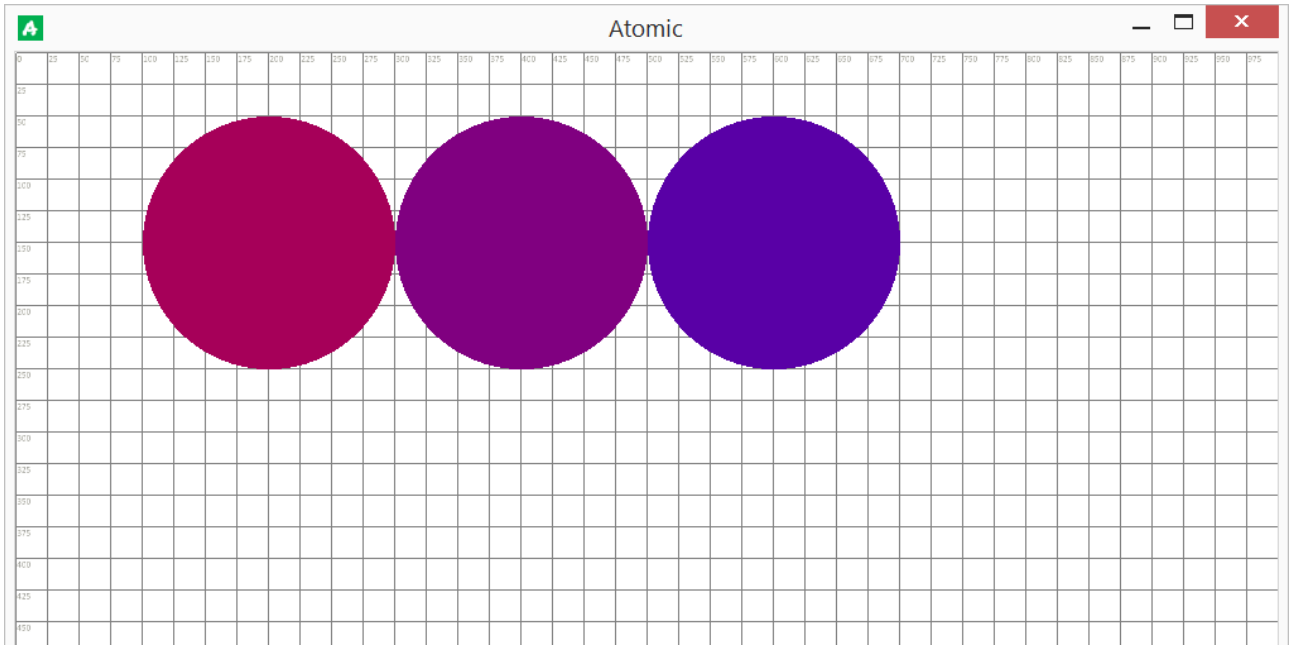
```
ottieni miscela colori da → (COLORE 1: ) (COLORE 2: ) (VALORE: )
```

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 colore1 = ottieni miscela colori da → (COLORE 1: rosso) (COLORE 2: blu) (VALORE: 0.35)
2 colore2 = ottieni miscela colori da → (COLORE 1: rosso) (COLORE 2: blu) (VALORE: 0.5)
3 colore3 = ottieni miscela colori da → (COLORE 1: rosso) (COLORE 2: blu) (VALORE: 0.65)
4 disegna cerchio → (X: 200) (COLORE: colore1)
5 disegna cerchio → (X: 400) (COLORE: colore2)
6 disegna cerchio → (X: 600) (COLORE: colore3)
```

Mescolando rosso e blu in proporzioni diverse ottengo varie tonalità di viola/fucsia.



Funzioni base 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

```
1  INIZIA
2  suona → (SUONO: suono bau) (VOLUME: 0.5) (INTONAZIONE: Fa diesis) (RIPETI SUONO: falso)
3
4  CICLO CONTINUO
```

Le costanti musicali possono essere utilizzate per intonare un suono.

Esempio

```
1  suona → (SUONO: suono miao) (INTONAZIONE: Fa diesis)
```

Di base un suono è relativamente intonato in Do.
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). Questo perché la frequenza "normale" del miagolio di un gatto è diversa ad esempio da quella del muggito di una mucca.

Esempio

```
1  correzione = La
2  suona → (SUONO: suono muu) (INTONAZIONE: correzione + Fa diesis)
3  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, per ottenere note delle ottave superiori:

```
<nota>*2^<ottava>
```

Per ottenere note delle ottave inferiori:

```
<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
```

Per utilizzare un suono locale, inserito nella cartella delle risorse di Atomic bisogna specificare l'indirizzo e il nome del file.

Esempio

```
INIZIA
suona → (SUONO: "suoni/esplosione.ogg")

CICLO CONTINUO
```

Non è strettamente necessario specificare l'estensione del file, l'estensione viene riconosciuta automaticamente da Atomic:

Esempio

```
INIZIA
suona → (SUONO: "suoni/esplosione")

CICLO CONTINUO
```

Attualmente sono supportati solo i suoni in formato ".ogg". È possibile convertire suoni in formato ".ogg" utilizzando vari servizi online, ad esempio https://online-audio-converter.com/it/.

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.".

Esempio

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

CICLO CONTINUO
```

Per un maggiore controllo dello streaming è disponibile la funzione **ottieni suono** per caricare il suono e memorizzarlo in una variabile. In questo modo anche se il file è pesante (ad esempio se è una musica o una canzone) il suono partirà senza ritardi.

Esempio

```
INIZIA
musica = ottieni suono →
(INDIRIZZO: "internet/upload.wikimedia.org/wikipedia/en/4/45/ACDC_-_Back_In_Black-sample.ogg")

CICLO CONTINUO
se tasto invio è stato premuto = vero allora suona → (SUONO: musica) .
```

Altre funzioni base sull'audio sono:

```
interrompi tutti i suoni
```

```
imposta volume principale → (VOLUME:)
```

1=massimo, 0=muto

```
ottieni volume principale
```

Funzioni sugli oggetti audio dinamico

Gli oggetti audio dinamico sono degli oggetti speciali che permettono di **gestire e modificare l'audio durante la sua esecuzione**.

Tramite la funzione *suona* è possibile riprodurre direttamente un suono ma non è possibile gestirlo durante la sua esecuzione: una soluzione cosi semplice può andare bene per riprodurre degli effetti sonori di breve durata ma non è adatta per gestire tracce sonore lunghe. Usando la funzione *suona audio dinamico* invece si può avere il pieno controllo di ogni singolo esemplare di suono in esecuzione.

Per comprendere l'audio dinamico è bene distinguere il suono inteso come risorsa dall'esemplare di un suono in riproduzione.

Il suono come risorsa può essere già integrato in Atomic (costanti sull'audio come "*suono bau*") o essere caricato esternamente (locale o dal web in **formato .ogg**) e rappresenta un modello di base di traccia audio.

Con esemplare di un suono s'intende un suono riprodotto durante l'esecuzione del codice tramite la funzione *suona audio dinamico*. Esso è collocato precisamente nel tempo e nello spazio tridimensionale (simulato) dell'ascoltatore e per questo è unico, identificabile e modificabile.

Le funzione dedicate agli oggetti audio dinamico sono le seguenti:

```
suona audio dinamico → (NOME:) (SUONO: ) (INOTNAZIONE:) (VOLUME:)  (X:) (Y:) (Z:)
                       (RIPETI SUONO:) (POSIZIONE TRACCIA:)
```

```
modifica audio dinamico → (NOME:) (INOTNAZIONE:) (VOLUME:) (X:) (Y:) (Z:) (POSIZIONE TRACCIA:)
```

```
ottieni risultato controllo se sta suonando → (NOME:)
```

```
metti in pausa questo suono → (NOME:)
```

```
riprendi a suonare questo suono → (NOME:)
```

```
ottieni risultato controllo se questo suono è in pausa → (NOME:)
```

```
ferma questo suono → (NOME:)
```

```
ottieni posizione attuale traccia del suono → (NOME:)
```

```
ottieni lunghezza traccia del suono → (NOME:)
```

```
modifica posizione ascoltatore → (X:) (Y:) (Z:)
```

Esempio

```
INIZIA
frequenza = 1
suona audio dinamico → (SUONO: "suoni/batteria.ogg") (NOME: batteria) (VOLUME: 0.5)
suona audio dinamico → (SUONO: "suoni/batteria_rock.ogg") (NOME: batteria_rock) (VOLUME: 0.5)
suona audio dinamico → (SUONO: "suoni/xilofono.ogg") (NOME: xilofono) (VOLUME: 0.5)
suona audio dinamico → (SUONO: "suoni/flauto.ogg") (NOME: flauto) (VOLUME: 0.5)
suona audio dinamico → (SUONO: "suoni/tromboni.ogg") (NOME: tromboni) (VOLUME: 0.5)
suona audio dinamico → (SUONO: "suoni/tuba.ogg") (NOME: tuba) (VOLUME: 0.5)
suona audio dinamico → (SUONO: "suoni/contrabbasso.ogg") (NOME: contrabbasso) (VOLUME: 0.5)
suona audio dinamico → (SUONO: "suoni/coro_ragazzi.ogg") (NOME: ragazzi) (VOLUME: 0.5)
suona audio dinamico → (SUONO: "suoni/coro_uomini.ogg") (NOME: uomini) (VOLUME: 0.5)
suona audio dinamico → (SUONO: "suoni/coro_principale.ogg") (NOME: misto) (VOLUME: 0.5)
suona audio dinamico → (SUONO: "suoni/basso_elettrico.ogg") (NOME: basso) (VOLUME: 0.5)
suona audio dinamico → (SUONO: "suoni/chitarra_1.ogg") (NOME: chitarra_1) (VOLUME: 0.5)
suona audio dinamico → (SUONO: "suoni/chitarra_2.ogg") (NOME: chitarra_2) (VOLUME: 0.5)

CICLO CONTINUO
modifica audio dinamico → (NOME: batteria) (INTONAZIONE: frequenza)
modifica audio dinamico → (NOME: batteria_rock) (INTONAZIONE: frequenza)
modifica audio dinamico → (NOME: xilofono) (INTONAZIONE: frequenza)
modifica audio dinamico → (NOME: flauto) (INTONAZIONE: frequenza)
modifica audio dinamico → (NOME: tromboni) (INTONAZIONE: frequenza)
modifica audio dinamico → (NOME: tuba) (INTONAZIONE: frequenza)
modifica audio dinamico → (NOME: contrabbasso) (INTONAZIONE: frequenza)
modifica audio dinamico → (NOME: ragazzi) (INTONAZIONE: frequenza)
modifica audio dinamico → (NOME: uomini) (INTONAZIONE: frequenza)
modifica audio dinamico → (NOME: misto) (INTONAZIONE: frequenza)
modifica audio dinamico → (NOME: basso) (INTONAZIONE: frequenza)
modifica audio dinamico → (NOME: chitarra_1) (VOLUME: 1) (INTONAZIONE: frequenza)
modifica audio dinamico → (NOME: chitarra_2) (VOLUME: 1) (INTONAZIONE: frequenza)

se tasto invio è premuto = vero {diminuisci frequenza di 0.01}
se tasto invio è premuto = falso e frequenza < 1 { aumenta frequenza di 0.05 }
frequenza = ottieni il valore limitato della variabile tra → (VALORE 1: 0)
                                                            (VALORE 2: 1)
                                                            (VARIABILE: frequenza)

posizione = ottieni posizione attuale traccia del suono → (NOME: batteria)
disegna testo → (TESTO: "Posizione traccia: <posizione> secondi (a capo)Frequenza: <frequenza>")
```

Questo esempio suona in contemporanea diverse tracce audio (strumenti e suoni diversi) che compongono una musica.
È possibile modificare in tempo reale il volume e l'intonazione per ogni traccia.

Comporre melodie

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

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

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

Esempio

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

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

Esempio

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

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

Esempio

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

```
INIZIA
//Fra Martino
suona melodia →
(SUONO: suono_nota_piano )
(NOTE: Do--Re--Mi--Do--Do--Re--Mi--Do--Mi--Fa--Sol----Mi--Fa--Sol----$
Sol-La-Sol-Fa-Mi--Do--Sol-La-Sol-Fa-Mi--Do--Do--Sol--Do----Do--Sol--Do-----)
(BPM: 200)
(RIPETI MELODIA: vero)

CICLO CONTINUO
```

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

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  crea tasto virtuale → (NOME: tasto) (ETICHETTA: "Cliccami!")
3                        (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: tasti_neri) (X: 600) (COLORE SFONDO: nero) (COLORE TESTO: giallo)
3                         (ALTEZZA: 50) (ETICHETTA: "Sono un tasto") (TRASPARENZA: 0.75)
4
5  crea tasto virtuale → (NOME: tasto_a) (CLASSE: tasti_neri) (Y: 50)
6  crea tasto virtuale → (NOME: tasto_b) (CLASSE: tasti_neri) (Y: 150)
7  crea tasto virtuale → (NOME: tasto_c) (CLASSE: tasti_neri) (Y: 250) (ETICHETTA: "abc")
8  crea tasto virtuale → (NOME: tasto_d) (CLASSE: tasti_neri) (Y: 350)
9
10 CICLO CONTINUO
```

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: "interuttore") (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:)
```

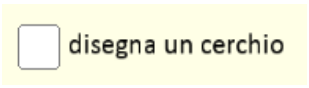
L'argomento ETICHETTA è una stringa di testo che viene disegnata a sinistra della casella, questa può essere diversa dal nome. L'argomento VALORE PREDEFINITO indica lo stato della casella al momento della sua creazione (0 o 1, vedi poco più sotto).

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

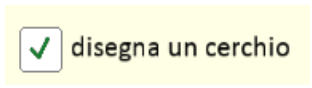
La variabile (che viene creata automaticamente assieme alla casella) contiene un valore booleano: 1 (vero, ON) o 0 (falso, OFF)

Esempio

```
1  INIZIA
2  crea casella di spunta → (NOME: impostazione) (VALORE PREDEFINITO: 0)
3                           (ETICHETTA: "disegna un cerchio") (X: 310) (Y: 210)
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

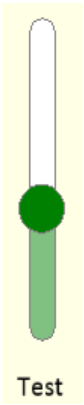
Le barre di controllo sono degli **oggetti** speciali disegnati all'interno della finestra; possono avere un colore personalizzato e un'etichetta descrittiva.

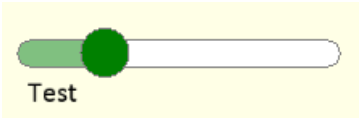
Le barre di controllo permettono di selezionare tramite il mouse un valore compreso tra un massimo e un minimo specificati.

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:)
```





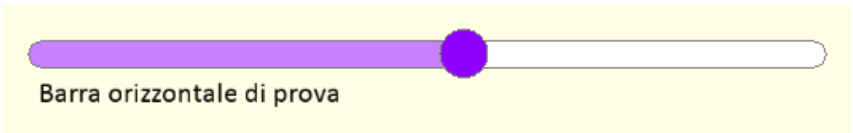
In base alla funzione utilizzata le barre possono avere una LARGHEZZA o un'ALTEZZA personalizzata.
L'argomento ETICHETTA è una stringa di testo che viene disegnata sotto la barra, questa può essere diversa dal nome.
L'argomento VALORE PREDEFINITO indica il valore della barra al momento della sua creazione.

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: 50) (NOME: barra) (VALORE MINIMO: 50)
3                                        (COLORE: viola) (LARGHEZZA: 500) (VALORE PREDEFINITO: 234)
4                                        (VALORE MASSIMO: 500) (ETICHETTA: "disegna un cerchio")
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).

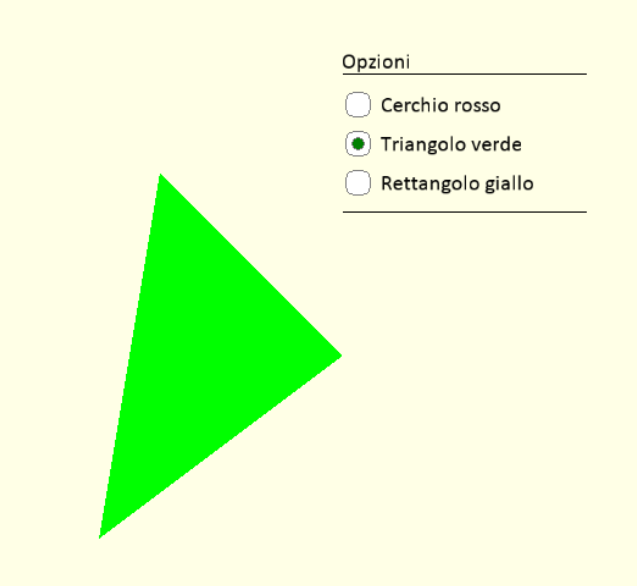
È possibile specificare fino a 8 valori (opzioni) per ogni gruppo; questi valori possono essere numeri o stringhe di testo.

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:)
```

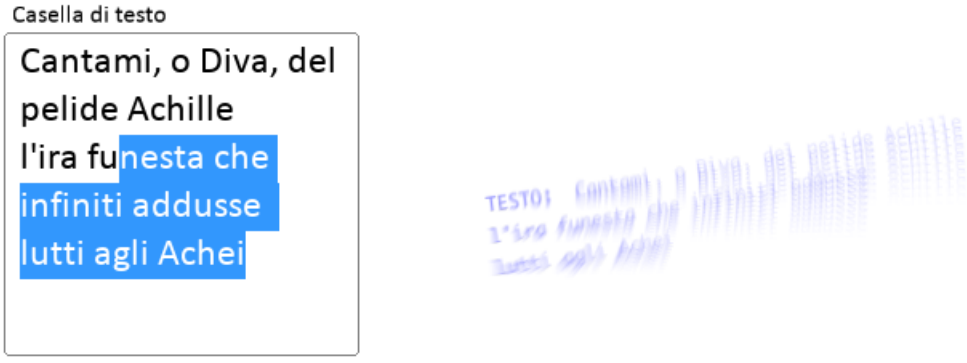
L'argomento ETICHETTA è una stringa di testo che viene disegnata sopra la casella, questa può essere diversa dal nome.
L'argomento TESTO indica il testo predefinito della casella al momento della sua creazione.
L'argomento COLORE indica il colore del testo.

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

```
1  INIZIA
2  imposta griglia → (VISIBILE: falso)
3  crea casella di testo multilinea → (NOME: casella) (X: 100) (Y: 100)
4                                     (ETICHETTA: "casella di testo")
5
6  CICLO CONTINUO
7  angolo = 0
8  ripeti per 10 volte
9  {
10 aumenta angolo di 1
11 disegna testo → (X: 400) (Y: 200) (TESTO: "TESTO: <casella>") (COLORE: blu)
12                 (ROTAZIONE: angolo) (TRASPARENZA: angolo/100)
13 }
```



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 tramite la funzione *imposta testo in una casella di testo*:

```
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: "prova")
}
```

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: 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
//obbiettivo: 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 = vero {crea tasto virtuale → (NOME: tasto2) }
//SBAGLIATO!!! Ogni volta che il tasto invio viene premuto viene creato un nuovo tasto

se tasto invio è stato premuto = vero e interfaccia  = "non è ancora stata creata"
{
crea tasto virtuale → (NOME: tasto2) (TRASPARENZA: 0.1)
interfaccia  = "è stata creata"
}
//OK, questo codice crea il tasto solo la prima volta che il tasto invio viene premuto
```

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  disegna testo → (TESTO: "la data di oggi è <data>")
```

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

```
nome_giorno = ottieni nome del giorno attuale
disegna testo → (TESTO: "Oggi è <nome_giorno>")
```

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

```
giorno = ottieni una data → (GIORNO: 6) (MESE: 10) (ANNO: 2025)
la_vefica_è_oggi = ottieni risultato controllo se la data è oggi → (DATA: giorno)
se la_vefica_è_oggi = vero { disegna testo → (TESTO: "Oggi ci sarà la verifica di storia") }
se la_vefica_è_oggi = falso { disegna testo → (TESTO: "Oggi non ci sarà la 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

```
oggi = ottieni data attuale
nome_mese = ottieni elemento da questa data → (DATA: oggi) (ELEMENTO: "nome del mese")
disegna testo → (TESTO: "Siamo nel mese di <nome_mese>")
```

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 per la Pixel Art

La pixel art è un'attività didattica che ultimamente va molto di moda nelle scuole, soprattutto nella primaria. Nonostante sia sicuramente un'attività interessante in tutte le sue forme, viene spesso proposta scorrettamente come attività di coding unplugged.
Usare un codice per disegnare un'immagine su un foglio di carta a quadretti non è coding ma l'esatto opposto: decoding, ovvero il lavoro che svolge automaticamente il computer quanto interpreta un codice.

Per fare una vera attività di coding con la pixel art bisognerebbe partire invece dal disegno. Una volta disegnata un'immagine su una griglia quadrettata bisogna riuscire a ricavarne il codice che la descrive. Questo codice, oltre a poter essere utilizzato per riprodurre il disegno su carta, può essere interpretato da Atomic ed in seguito può essere modificato e ampliato.

La funzione per disegnare un'immagine con la tecnica della pixel art è ***disegna pixel art***:

```
disegna pixel art → (DISEGNO:) (X:) (Y:) (LARGHEZZA:) (ALTEZZA:) (SCALA:) (ROTAZIONE:)
                    (TRASPARENZA:) (COLORE A:) (COLORE B:) (COLORE C:) … (COLORE P:)
```

L'argomento **DISEGNO** definisce il disegno tramite una stringa di testo (vedi più avanti).

Gli argomenti **X** e **Y** definiscono la posizione in cui verrà disegnata l'immagine.

Gli argomenti **LARGHEZZA** e **ALTEZZA** definiscono il numero di pixel che comporranno l'immagine e la loro organizzazione nello spazio (quante righe/colonne). Se questi argomenti non vengono specificati la dimensione dell'immagine sarà 10x10.

L'argomento **SCALA** definisce la dimensione dei pixel che compongono l'immagine e di conseguenza la scala di tutta l'immagine. Se l'argomento SCALA non viene specificato viene impostato il valore di base 25 (che corrisponde al valore di base della griglia di sfondo di Atomic). Se viene utilizzato il valore 1 (scala 1:1) la dimensione dei pixel corrisponderà alla dimensione dei pixel dello schermo e l'immagine risulterà molto più piccola ma definita.

L'argomento **ROTAZIONE** permette di ruotare l'immagine dalla sua origine (l'angolo in alto a sinistra).

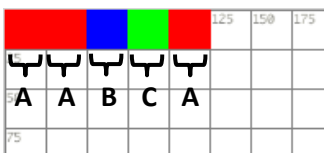
L'argomento **TRASPARENZA** permette di definire l'opacità del disegno.

Gli argomenti **COLORE A**, **COLORE B**, **COLORE C**, ecc. fino al **COLORE P** permettono di definire i vari colori utilizzati per rappresentare l'immagine. Per ogni immagine si possono definire fino a 16 colori (da A a P). Queste lettere vengono utilizzate nel codice che descrive il disegno.

Esempio

```
1  disegna pixel art → (DISEGNO: "AABCA") (COLORE A: rosso) (COLORE B: blu) (COLORE C: verde)
```

Il modo più semplice di definire un DISEGNO è scrivere una serie di lettere in una stringa di testo (es. "AABCA") dove ogni lettera rappresenta il colore di un pixel partendo dall'origine (in alto a sinistra).
Nell'esempio A definisce i pixel rossi, B i pixel blu e C i pixel verdi.

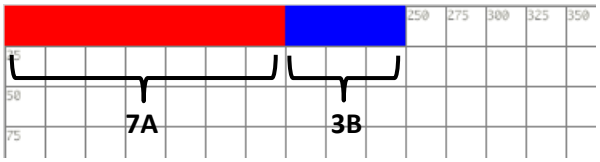


Il codice che definisce il disegno utilizza un algoritmo di tipo RLE (Run-length encoding, ovvero un algoritmo che disegna “correndo” lungo la larghezza dell’immagine). Ciò significa che non bisogna necessariamente definire il colore di ogni singolo pixel ma è possibile definire il colore di più pixel consecutivi con la sintassi numero-lettera "nX nX …".

Esempio

```
1  disegna pixel art → (DISEGNO: "7A 3B") (COLORE A: rosso) (COLORE B: blu)
```

Nell’esempio i primi 7 pixel del disegno saranno rossi e i successivi 3 saranno blu.

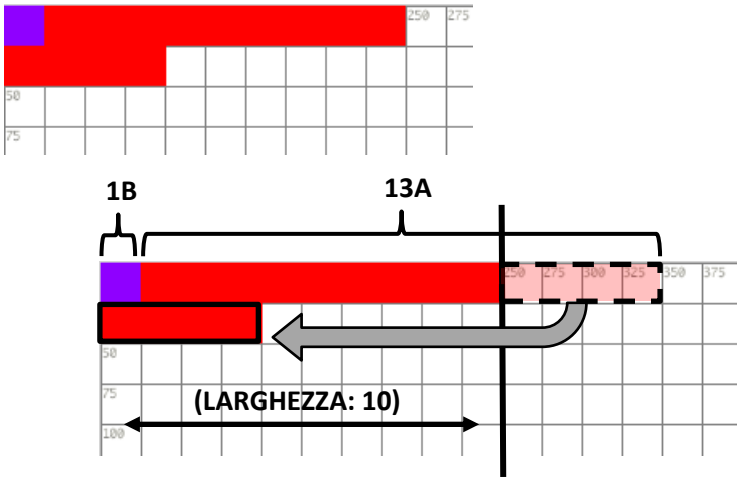


Gli spazi vuoti all’interno della stringa non influiscono sul risultato (ad esempio è indifferente scrivere "7A 3B", "7A3B" o "7 A 3 B"). Si può anche mandare a capo il codice all’interno della stringa; anche questa azione non influisce sul risultato.

I pixel consecutivi che teoricamente andrebbero posizionati oltre la larghezza dell’immagine vengono riportati automaticamente all’inizio della riga successiva.

Esempio

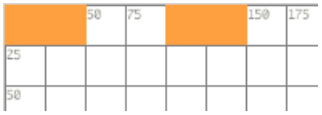
```
1  disegna pixel art → (LARGHEZZA: 10) (ALTEZZA: 10) (DISEGNO: "1B 13A") (COLORE A: rosso) (COLORE B: viola)
```



I pixel di cui non viene specificato il colore rimarranno trasparenti (non verranno disegnati).
È anche possibile utilizzare la parola chiave *trasparente* come valore degli argomenti COLORE *X*.

Esempio

```
1  disegna pixel art → (DISEGNO: "2A 2B 2A") (COLORE A: arancione) (COLORE B: trasparente)
```



Oltre a specificare valori numerici è anche possibile **utilizzare delle espressioni per indicare le ripetizioni dei pixel**.

Esempio

```
disegna pixel art → (DISEGNO: "<x del mouse/100>A 5E") (COLORE A: azzurro) (COLORE E: rosa)
```

In questo esempio la ripetizione dei pixel azzurri varia in base alla posizione orizzontale del mouse.

È anche possibile utilizzare direttamente una variabile che contiene una stringa di testo come valore dell'argomento DISEGNO o concatenare in una stringa più variabili che contengono testo come valore.

Esempio

```
codice = "3C"
se tasto barra spaziatrice è premuto = vero { codice = "<x del mouse/100>A 4B" }
disegna pixel art → (DISEGNO: codice) (COLORE A: azzurro) (COLORE B: verde) (COLORE C: giallo)
```

Esempio

```
codice_1 = "3A 4B"
codice_2 = "7C"
se tasto invio è premuto = vero { codice_2 = "4A" }
disegna pixel art → (DISEGNO: "<codice_1><codice_2>")
                    (COLORE A: azzurro) (COLORE B: verde) (COLORE C: giallo)
```

Usando variabili ed espressioni è pertanto possibile **modificare e animare le pixel art**.

Per ottenere effetti di **animazione del colore** si possono utilizzare anche gli argomenti COLORE *X*.

Esempio

```
INIZIA
tinta = 0

CICLO CONTINUO
aumenta tinta di 1
colore_animato = ottieni colore hsv → (TINTA: tinta)
disegna pixel art → (DISEGNO: "2A 2B 2A") (COLORE A: azzurro) (COLORE B: colore_animato)
```

In questo esempio i pixel "B" cambiano automaticamente colore nel tempo

Esempio completo

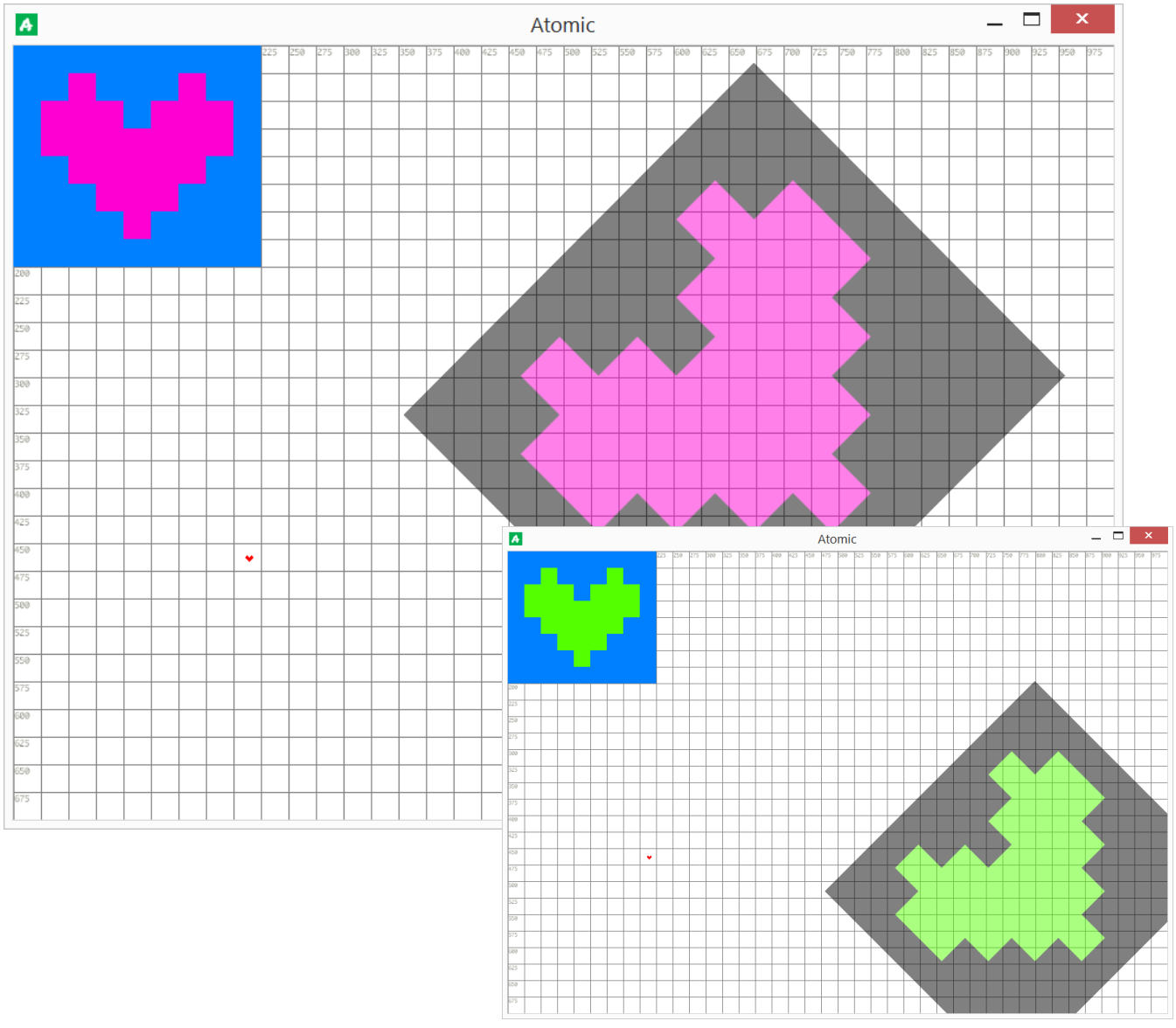
```
INIZIA
tinta = 0

CICLO CONTINUO
aumenta tinta di 1
colore_animato = ottieni colore hsv ➔ (TINTA: tinta)
disegno_cuore = "11A B 3A B 3A 3B A 3B 2A 7B 3A 5B 5A 3B 7A B 13A"

//disegno cuore base (animato)
disegna pixel art ➔ (DISEGNO: disegno_cuore) (LARGHEZZA: 9) (ALTEZZA: 8)
                    (COLORE A: azzurro) (COLORE B: colore_animato)

//disegno cuore modificato
disegna pixel art ➔ (DISEGNO: disegno_cuore) (LARGHEZZA: 9) (ALTEZZA: 8)
                    (COLORE A: nero) (COLORE B: colore_animato)
                    (X: x del mouse) (Y: y del mouse) (ROTAZIONE: 45)
                    (TRASPARENZA: 0.5) (SCALA: 50)

//disegno cuore in scala 1:1
disegna pixel art ➔ (DISEGNO: disegno_cuore) (LARGHEZZA: 9) (ALTEZZA: 8)
                    (COLORE A: bianco) (COLORE B: rosso)
                    (X: 210) (Y: 460) (SCALA: 1)
```



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:)
```

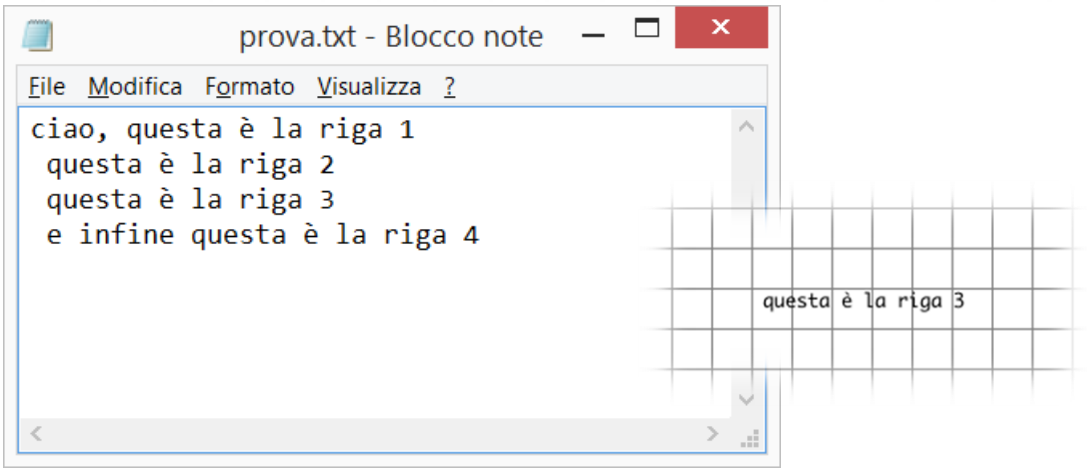
Con questa funzione è possibile ottenere la riga di testo specificata del file.

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

```
1  nome_giocatore = "Luigi"
2  punteggio = 425248
3  scrivi una nuova linea su questo file → (NOME: "classifica.txt")
4                                          (TESTO: "<nome_giocatore> : <punteggio>")
```

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:

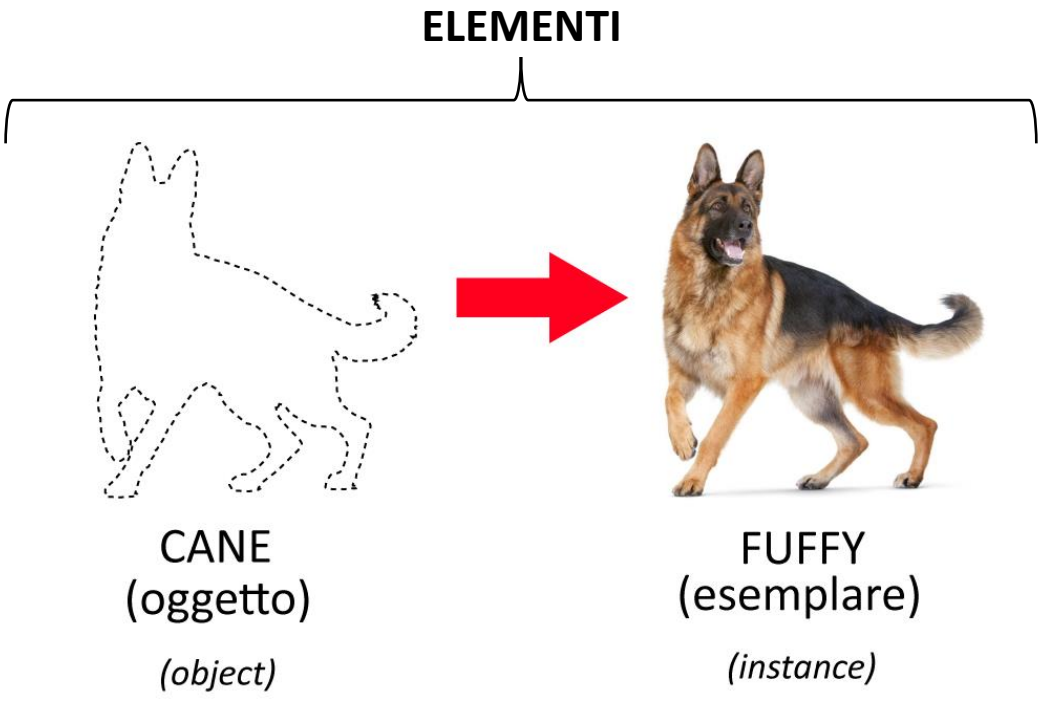
Oggetto: *cane*

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.

ESEMPIO CONCRETO

A cosa serve concretamente tutto questo sistema? Immaginate un videogioco di qualsiasi genere in cui c'è un giocatore e vari tipi di nemici da sconfiggere. Il giocatore (l'oggetto che controlla il giocatore) è uno solo , quindi "giocatore" può essere programmato come un esemplare unico. I nemici invece sono tanti e di vario tipo. I nemici hanno una parte del loro comportamento in cumune e delle caratteristiche peculiari per ogni tipo; per questo motivo una buona strategia per programmare i nemici è creare un oggetto generico "nemico" per descrivere le caratteritiche e il comportamento comune per tutti i nemici per poi definire tutti i tipi di avversi, assegnando "nemico" come loro oggetto genitore e defininendo poi nel dettaglio le loro peculiarità. Infine una volta ottenuti i vari modelli generici dei nemici si decide dove, quando e quanti nemici saranno presenti "fisicamente" nel gioco definendo la creazione degli esemplari. Questo esempio riassume gran parte della logica riguardante i concetti di riuso del codice, modularità e manutenibilità tipici della programmazione ad oggetti.

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]
VELOCITA ANIMAZIONE: indica la velocità d'animazione dell'immagine in fotogrammi al secondo [default: 0]
FOTOGRAMMA: indica Il fotogramma corrente dell'animazione [default: 0]
TIMER 1, **TIMER 2**, **TIMER 3**, **TIMER 4**: indicano i timer locali in secondi [default: 0]
DURATA: indica la durata di vita in secondi dell'esemplare [default: ∞]

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

```
crea un oggetto → (NOME: freccia) // Nome delloggetto
                  (X: 20) (Y: 20) // Coordinate in cui verranno creati i suoi esemplari
                  (DIREZIONE: -45) //rotazione dell'immagine
                  (ROTAZIONE: -45) // Direzione di movimento
                  (VELOCITA: 3) // Velocità di movimento in pixel per step
                  (IMMAGINE: "immagini/freccia") // L'immagine 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**.

Esempio

```
crea un esemplare → (OGGETTO: freccia)
crea un esemplare → (OGGETTO: freccia) (COLORE: verde)
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) (benzina: 100) (colore_carrozzeria: blu)
```

Gli oggetti in Atomic sono predisposti per essere rappresentati graficamente e in modo automatico come esemplari; questa caratteritica è molto comoda per realizzare applicazioni multimediali e videogiochi ma non sempre è utile: gli oggetti possono anche essere usati come strutture di dati generiche.

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.

! Esempio

```
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'elemento.

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

La forma da utilizzare è:

```
(ETICHETTA: VARIABILE LOCALE)
```

Esempio

```
1  modifica un elemento → (NOME: missile) (DIREZIONE: direzione) (ROTAZIONE: DIREZIONE)
2  //L'angolo 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 = vero allora modifica un elemento → (NOME: cane) (X: X+3) .
2  se tasto freccia sinistra è premuto = vero allora modifica un elemento → (NOME: cane) (X: X-3) .
3  se tasto freccia su è premuto = vero allora modifica un elemento → (NOME: cane) (Y: Y-3) .
4  se tasto freccia giu è premuto = vero allora modifica un elemento → (NOME: cane) (Y: Y+3) .
5  //premendo le frecce della tastiera il cane si sposta della sua coordinata attuale + 3
```

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

Utilizzando il costrutto "**del"** è possibile leggere ed utilizzare una variabile locale all'interno di qualsiasi espressione.

La forma da utilizzzare è:

```
<nome variabile> del <nome esemplare>
```

Esempio

```
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 di un esemplare ma non di modificarla.
L'unico modo per modificare dall'esterno una variabile locale è utilizzando la funzione modifca elemento.

Ad esempio, non è possbile scrivere:

Esempio

```
X del cavaliere = 500 //<-- SBAGLIATO!!
```

La forma corretta da utilizzare è la seguente:

Esempio

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

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

Esempio

```
x = VELOCITA dell'auto
y = ROTAZIONE della palla
z = peso dello zucchero
```

Il costrutto "del" può essere utilizzato precedendo il nome di un esemplare ma non di un oggetto. È tuttavia possibile utilizzarlo su un insieme di esemplari, sfruttando una variabile per memorizzare dinamicamente i nomi degli esemplari di cui si vogliono leggere le variabili locali. È possibile utilizzarlo anche all'interno di un'espressione in una stringa di testo.

Esempio

```
se tasto invio è premuto = vero
{
x = ottieni un valore intero compreso tra questi → (VALORE 1: 0) (VALORE 2: larghezza finestra)
y = ottieni un valore intero compreso tra questi → (VALORE 1: 0) (VALORE 2: altezza finestra)
crea un esemplare → (IMMAGINE: "immagini/pallina") (X: x) (Y: y)
}

esemplare_cliccato = ottieni ultimo esemplare cliccato
disegna testo → (TESTO: "Coordinate: (<X dell'esemplare_cliccato>,<Y dell'esemplare_cliccato>)")
```

Funzioni per muovere gli esemplari

`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

```
1  //Muovi il giocatore verso il cursore quando il tasto sinistro del mouse è premuto
2  se tasto sinistro del mouse è premuto = vero
3  {
4  muovi un elemento verso un punto → (ELEMENTO: giocatore) (VELOCITA: 10)
5                                     (X: x del mouse) (Y: y del mouse)
6  }
```

`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

```
1  //Teletrasporta la palla al lato opposto quando esce dalla finestra
2  trasporta elemento al lato opposto quando esce dalla finestra → (ELEMENTO: palla)
```

Funzioni per ottenere informazioni sugli esemplari

```
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. Le collisioni funzionano solo se gli esemplari coinvolti hanno un'immagine associata tramite l'argomento **IMMAGINE**.

Esempio

```
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

```
//Controlla se i nemici sono stati abbattuti: se si fai apparire tre nuovi nemici in una posizione casua
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)
  }
}
```

```
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

```
//L'obbiettivo 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

```
1  //L'obbiettivo del missile è il nemico più lontano
2  obbiettivo = ottieni nome dell'esemplare più lontano da → (X: X del missile) (Y: Y del missile)
3                                                           (OGGETTO: nemico)
```

```
ottieni numero esemplari esistenti di → (OGGETTO:)
```

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

Esempio

```
1  //Controlla se ci sono meno di 4 nemici nella finestra:
2  //se si fai apparire tre nuovi nemici in una posizione casuale
3  nemici_presenti = ottieni numero esemplari esistenti di → (OGGETTO: nemico)
4  se nemici_presenti < 4
5  {
6    ripeti per 3 volte
7    {
8    x_casuale = ottieni un valore compreso tra questi → (VALORE 1: 0) (VALORE 2: larghezza finestra)
9    y_casuale = ottieni un valore compreso tra questi → (VALORE 1: 0) (VALORE 2: altezza finestra)
10   crea un esemplare → (OGGETTO: nemico) (X: x_casuale) (Y: y_casuale)
11   }
12 }
```

```
ottieni numero totale di esemplari esistenti
```

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

Esempio

```
1  //Controlla il numero complessivo di tutti gli esemplari nella finestra:
2  tot = ottieni numero totale di esemplari esistenti
3  disegna testo → (TESTO: "Ci sono <tot> esemplari nella finestra") (X: 10) (Y: 10)
```

```
ottieni risultato controllo se elemento è stato cliccato → (ELEMENTO:) (TASTO:)
```

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*). L'argomento **TASTO** indica il tasto del mouse da controllare: "sinistro", "destro", "centrale".
Se l'argomento **TASTO** non viene specificato viene controllato il tasto sinistro.

Esempio

```
1 //Se viene cliccata la palla diventa rossa
2 palla_cliccata = ottieni risultato controllo se elemento è stato cliccato → (ELEMENTO: palla)
3 se palla_cliccata = vero allora modifica un elemento → (OGGETTO: palla) (COLORE: rosso) .
```

```
ottieni risultato controllo se elemento è cliccato → (ELEMENTO:) (TASTO:)
```

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*). L'argomento **TASTO** indica il tasto del mouse da controllare: "sinistro", "destro", "centrale".
Se l'argomento **TASTO** non viene specificato viene controllato il tasto sinistro.

Esempio

```
1 //trascina la palla con il mouse
2 trascina = ottieni risultato controllo se elemento è cliccato → (ELEMENTO: palla)
3 se trascina = vero allora modifica un elemento → (NOME: palla) (X: x del mouse) (Y: y del mouse) .
```

```
ottieni ultimo esemplare cliccato → (ELEMENTO:) (TASTO:)
```

Ottieni un riferimento univoco all'ultimo esemplare cliccato con il tasto del mouse specificato tramite l'argomento **TASTO** ("sinistro", "destro", o "centrale"). Se l'argomento **TASTO** non viene specificato viene controllato il tasto sinistro.

Esempio

```
1 //Quando clicchi una moneta con il tasto destro aumenta il punteggio in base al suo valore
2 click = ottieni risultato controllo se elemento è cliccato  (ELEMENTO: moneta) (TASTO: "destro")
3 moneta_cliccata = ottieni ultimo esemplare cliccato → (TASTO: "destro")
4
5 se click = vero { aumenta punteggio di VALORE della moneta_cliccata }
```

```
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

```
1  //Se il mouse entra nella palla, la palla diventa verde
2  mouse_dentro = ottieni risultato controllo se il mouse è entrato in questo elemento → (ELEMENTO: palla)
3  se mouse_dentro = vero allora modifica un elemento → (NOME: 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

```
1  //Se il mouse esce dalla palla, la palla diventa gialla
2  mouse_uscito = ottieni risultato controllo se il mouse è uscito da questo elemento → (ELEMENTO: palla)
3  se mouse_uscito = vero allora modifica un elemento → (NOME: palla) (COLORE: giallo) .
```

```
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

```
1  //Riposiziona la palla al centro se esce dalla finestra
2  palla_fuori = ottieni risultato controllo se elemento è uscito dalla finestra → (ELEMENTO: palla)
3  se palla_fuori = vero allora modifica un elemento → (OGGETTO: palla) (X: larghezza finestra/2)
4                                                                      (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

```
//Fai tornare indietro la palla quando tocca il bordo della finestra
bordo = ottieni risultato controllo se elemento ha toccato il bordo della finestra → (ELEMENTO: palla)
se bordo = vero allora modifica un elemento → (OGGETTO: palla) (DIREZIONE: DIREZIONE-180)
```

`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

```
//Inserisci una moneta nel gioco in una posizione casuale

//Ottieni una posizone 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 casuale è libera o occupata
posizione_occupata = ottieni risultato controllo se ci sono elementi in questo punto → (X: x_casuale)
                                                                                       (Y: y_casuale)
//Se la posizione non è occupata da altri elementi allora crea la moneta
se posione_occupata = falso { crea un esemplare → (OGGETTO: moneta) (X: x casuale) (Y: y_casuale) }
```

`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

```
//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) (VELOCITA: 10)
                                   (X: X del giocatore) (Y: Y del giocatore)
}
```

Funzioni per distruggere gli esemplari

```
distruggi un esemplare → (ESEMPLARE:)
```

Distrugge l'esemplare specificato.

Esempio

```
1  se vita = 0 allora distruggi un esemplare → (ESEMPLARE: giocatore)
```

```
distruggi tutti gli esemplari di → (OGGETTO:)
```

Distrugge tutti gli esemplari dell'oggetto specificato.

Esempio

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

```
distruggi elemento quando collide con → (ELEMENTO:) (ELEMENTO COINVOLTO:)
```

Distrugge l'elemento specificato tramite l'argomento **ELEMENTO** quando collide con l'elemento coinvolto nella collisione (esemplari di un oggetto o singolo esemplare) specificato tramite l'argomento **ELEMENTO COINVOLTO**.

Esempio

```
1  //Distruggi la navicella quando collide con l'asteroide
2  distruggi elemento quando collide con → (ELEMENTO: navicella) (ELEMENTO COINVOLTO: asteroide)
```

```
distruggi questi elementi quando collidono tra loro → (ELEMENTO 1: ) (ELEMENTO 2:)
```

Distrugge entrambi gli elementi specificati (esemplari di un oggetto o singoli esemplari) quando collidono tra loro.

Esempio

```
1  //Distruggi l'asteroide e il proiettile quando collidono tra loro
2  distruggi elemento quando collide con → (ELEMENTO: proiettile) (ELEMENTO COINVOLTO: asteroide)
```

```
distruggi elementi che si trovano in questo punto → (X: ) (Y: )
```

Distrugge tutti gli elementi le cui coordinate coincidono con il punto specificato.

Esempio

```
1  //distruggi gli elmenti che finiscono nella trappola
2  distruggi elementi che si trovano in questo punto → (X: X della trappola) (Y: Y della trappola)
```

```
distruggi elementi che si trovano in quest'area rettangolare → (ELEMENTO:) (X:) (Y:)
                                                               (BASE:) (ALTEZZA:)
```

Distrugge gli esemplari specificati tramite l'argomento **ELEMENTO** che si trovano all'interno dell'area rettangolare specificata dagli argomenti **X**, **Y**, **BASE** e **ALTEZZA**.

Esempio

```
1  //distruggi i nemici che si trovano all'interno della trappola quando premi il tasto invio
2  se tasto invio è premuto = vero
3  {
4  distruggi elementi che si trovano in quest'area rettangolare → (ELEMENTO: nemico)
5                                                                 (X: x_trappola) (Y: y_trappola)
6                                                                 (BASE: base_trappola)
7                                                                 (ALTEZZA: altezza_trappola)
8  }
```

```
distruggi elementi che si trovano in quest'area circolare → (ELEMENTO:) (X:) (Y:) (RAGGIO:)
```

Distrugge gli esemplari specificati tramite l'argomento **ELEMENTO** che si trovano all'interno dell'area circolare specificata dagli argomenti **X**, **Y** e **RAGGIO**.

Esempio

```
1  //distruggi i nemici che si trovano all'interno della trappola quando premi il tasto invio
2  se tasto invio è premuto = vero
3  {
4  distruggi elementi che si trovano in quest'area circolare → (ELEMENTO: nemico)
5                                                              (X: x_trappola) (Y: y_trappola)
6                                                              (RAGGIO: raggio_trappola)
7  }
```

```
distruggi elemento quando esce dalla finestra → (ELEMENTO:)
```

Distrugge l'elemento specificato quando esce dalla finestra.

Esempio

```
1  //distruggi il proiettile quando esce dalla finestra
2  //se il proiettile non venisse distrutto anche se non si vede continuerebbe ad esistere!
3  distruggi elemento quando esce dalla finestra → (ELEMENTO: proiettile)
```

```
distruggi elemento quando questa sua variabile locale equivale a → (ELEMENTO:) (VARIABILE:)
                                                                    (VALORE:)
```

Distrugge l'elemento specificato quando la variabile locale specificata (predefinita o personalizzata) contiene il valore specificato.

Esempio

```
1  //distruggi una pallina quando il suo tempo (impostato nel TIMER 1) è scaduto (0)
2  distruggi elemento quando questa sua variabile locale equivale a → (ELEMENTO: pallina)
3                                                                    (VARIABILE: TIMER 1)
4                                                                    (VALORE: 0)
```

Disegnare con gli oggetti

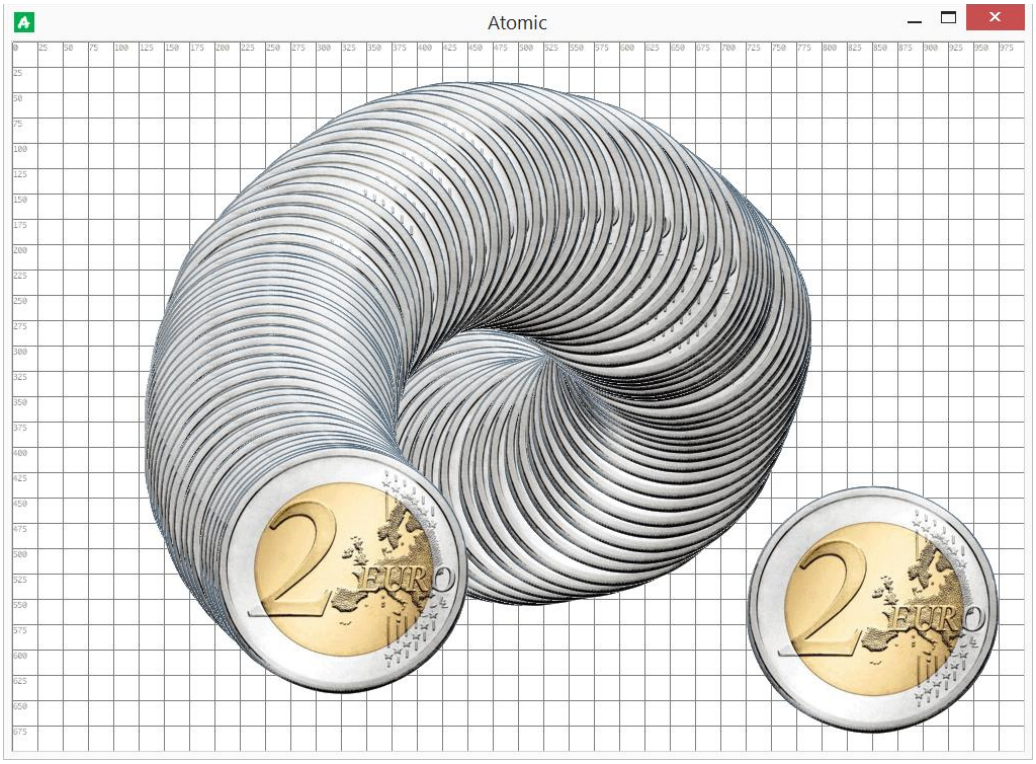
È possibile “stampare” nella finestra l’immagine attuale di uno o più esemplari tramite la funzione “timbra con questo elemento”.

```
timbra con questo elemento → (ELEMENTO: )
```

Esempio

```
1  INIZIA
2  crea un esemplare → (NOME: pennello) (IMMAGINE: "immagini/euro/2") (Z: -999)
3
4  CICLO CONTINUO
5  modifica un elemento → (NOME: pennello) (X: x del mouse) (Y: y del mouse)
6  se tasto sinistro del mouse è premuto = vero { timbra con questo elemento → (ELEMENTO: pennello) }
```

Questa funzione molto versatile rappresenta un’alternativa per disegnare immagini e creare disegni, forme e percorsi.



Per cancellare tutti i timbri esiste la funzione “cancella timbri”

```
Cancella timbri
```

Esempio

```
1  se tasto invio è stato premuto = vero allora cancella timbri .
```

Funzioni sui percorsi

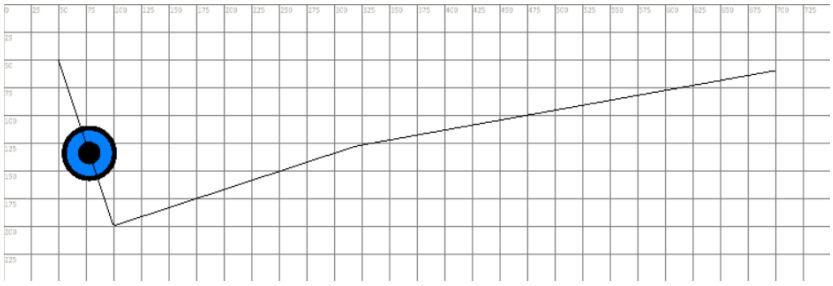
I percorsi sono degli insiemi di coordinate; sono delle risorse che vanno immagazzinate nelle variabili.

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

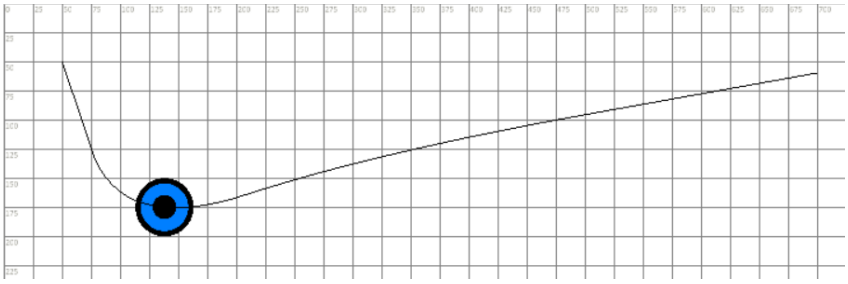
Per creare un percorso bisogna utilizzare la funzione *ottieni un percorso*.

```
ottieni un percorso → (TIPO:) (CHIUSO:) (COORDINATE: x,y / x,y / …,… )
```

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

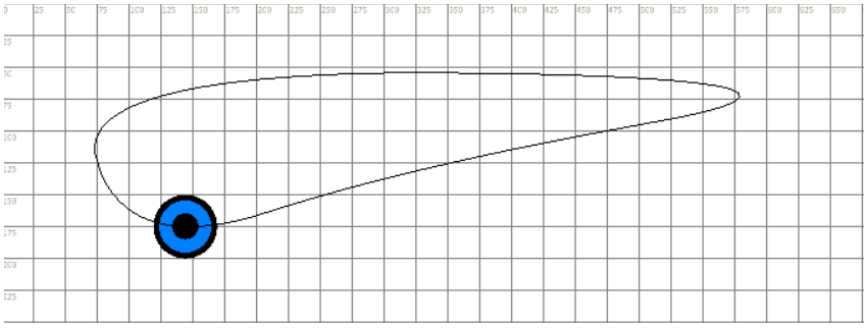


DRITTO

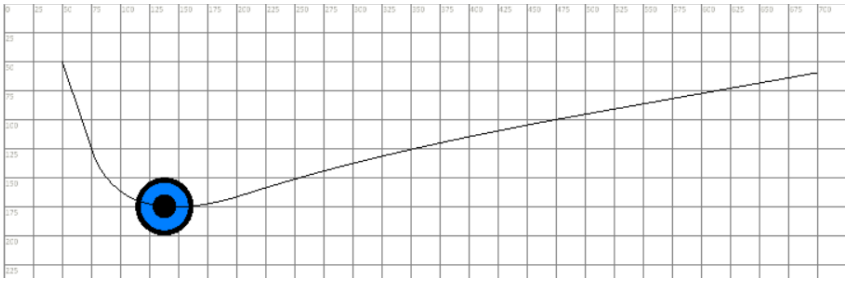


CURVE

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

```
percorso = ottieni un percorso → (TIPO: "curve") (CHIUSO: vero)
                                 (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

```
INIZIA
percorso = ottieni un percorso → (COORDINATE: 10,10 / 400,700 / 1000,500 )
                                 (TIPO: "curve") (CHIUSO: falso)

crea un esemplare → (NOME: freccia) (IMMAGINE: "immagini/freccia")

CICLO CONTINUO
//la freccia inizia a muoversi quando premi il tasto invio
se tasto invio è stato premuto = vero
{
assegna un percorso da seguire a un elemento → (ELEMENTO: freccia) (VELOCITA: 10)
                                               (PERCORSO: percorso)
                                               (QUANDO FINISCE: "torna indietro")
                                               (DISEGNA PERCORSO: vero)

modifica un elemento → (NOME: freccia) (ROTAZIONE: DIREZIONE)
}
```

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:)
```

Modificare un esemplare dall'interno

Normalmente il codice viene usato da un punto di vista globale ed esterno agli esemplari. Questo approccio va bene per i progetti più semplici ma in certi casi è limitante. Un esemplare di oggetto oltre a poter essere manipolato esternamente può essere **programmato dall'interno** tramite il costrutto "**esegue al suo interno questo codice**":

```
nome_elemento esegue al suo interno questo codice
{
…
}
```

Il codice eseguito tra le graffe dopo il costrutto si comporta diversamente rispetto al resto del codice: il codice è locale (soggettivo) e viene eseguito solo dagli elementi specificati, mediante il loro **punto di vista**. All'interno del costrutto è possibile leggere e modificare le caratteristiche predefinite come se fossero delle normali variabili. Anche le variabili locali personalizzate specificate durante la creazione dell'elemento possono essere utilizzate allo stesso modo.

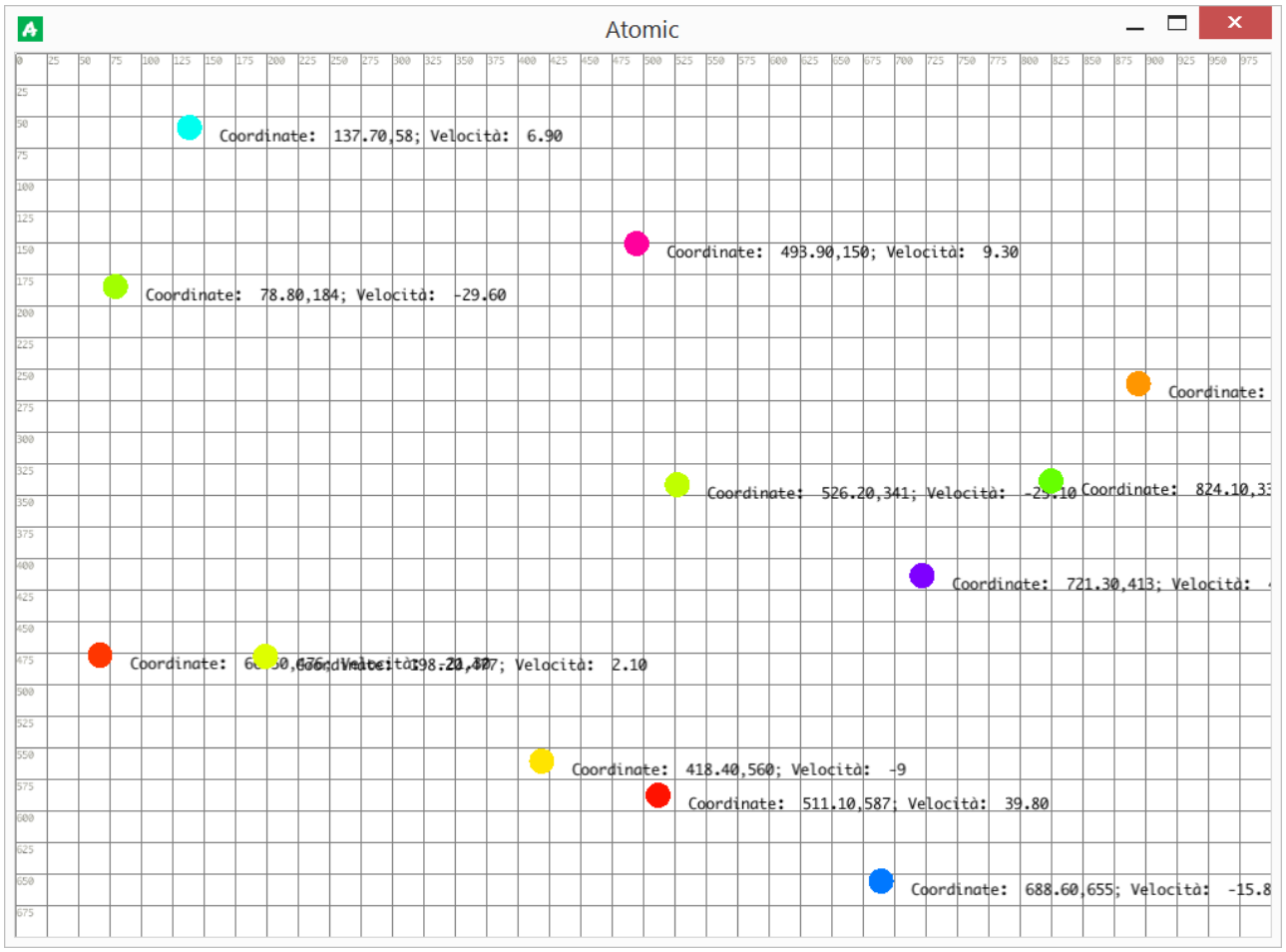
Le variabili inizializzate all'interno del costrutto "esegue al suo interno questo codice" vengono create come nuove variabili locali dell'elemento.

Anche se sembra un costrutto banale in realtà è molto potente, perché, oltre all'esecuzione del codice principale, permette di eseguire più codici parallelamente: uno per ogni esemplare di oggetto implicato. Questo costrutto espande di molto le potenzialità del linguaggio, perché tutti i costrutti e le funzioni di Atomic sono utilizzabili al suo interno.

Esempio

```
1  INIZIA
2  crea un oggetto → (NOME: pallina) (valore: 10) (tinta: 0)
3
4  CICLO CONTINUO
5  se tasto sinistro del mouse è stato premuto
6  {
7  crea un esemplare → (OGGETTO: pallina) (X: x del mouse) (Y: y del mouse)
8  }
9
10 pallina esegue al suo interno questo codice
11 {
12 aumenta X di valore
13 fattore_casuale = ottieni un valore compreso tra questi → (VALORE 1: 0.1) (VALORE 2: 5)
14 se X > larghezza finestra { valore = -10*fattore_casuale }
15 se X < 0 { valore = 10*fattore_casuale }
16 aumenta tinta di 1
17 COLORE = ottieni colore hsv → (TINTA: tinta)
18 disegna testo → (X: X+25) (Y: Y) (TESTO: "Coordinate: <X>,<Y>; Velocità: <valore>")
19 disegna cerchio → (X: X) (Y: Y) (COLORE: COLORE) (RAGGIO: 10)
20 }
```

Questo esempio crea tramite il mouse delle palline, completamente autonome, che cambiano colore e rimbalzano orizzontalmente a velocità diverse: senza il costrutto "esegue al suo interno questo codice" sarebbe impossibile ottenere lo stesso risultato. Da notare che all'oggetto non è associata nessuna immagine: la pallina viene disegnata tramite la funzione "disegna cerchio". Ogni esemplare disegna anche le sue coordinate e la sua velocità.



All'interno del costrutto "esegue al suo interno questo codice" un esemplare può riferirsi a sé stesso utilizzando la keyword "**sé stesso**". Questa keyword è utile soprattutto da utilizzare come valore per gli argomenti delle funzioni dedicate agli oggetti.

Esempio

```
1  pallina esegue al suo interno questo codice
2  {
3  collide_con_il_muro = ottieni risultato controllo collisione tra → (ELEMENTO 1: sé stesso)
4                                                                    (ELEMENTO 2: muro)
5
6  se collide_con_il_muro { distruggi un esemplare → (ESEMPLARE: sé stesso) }
7  }
```

L'esempio distrugge gli esemplari di pallina che collidono con il muro.

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.

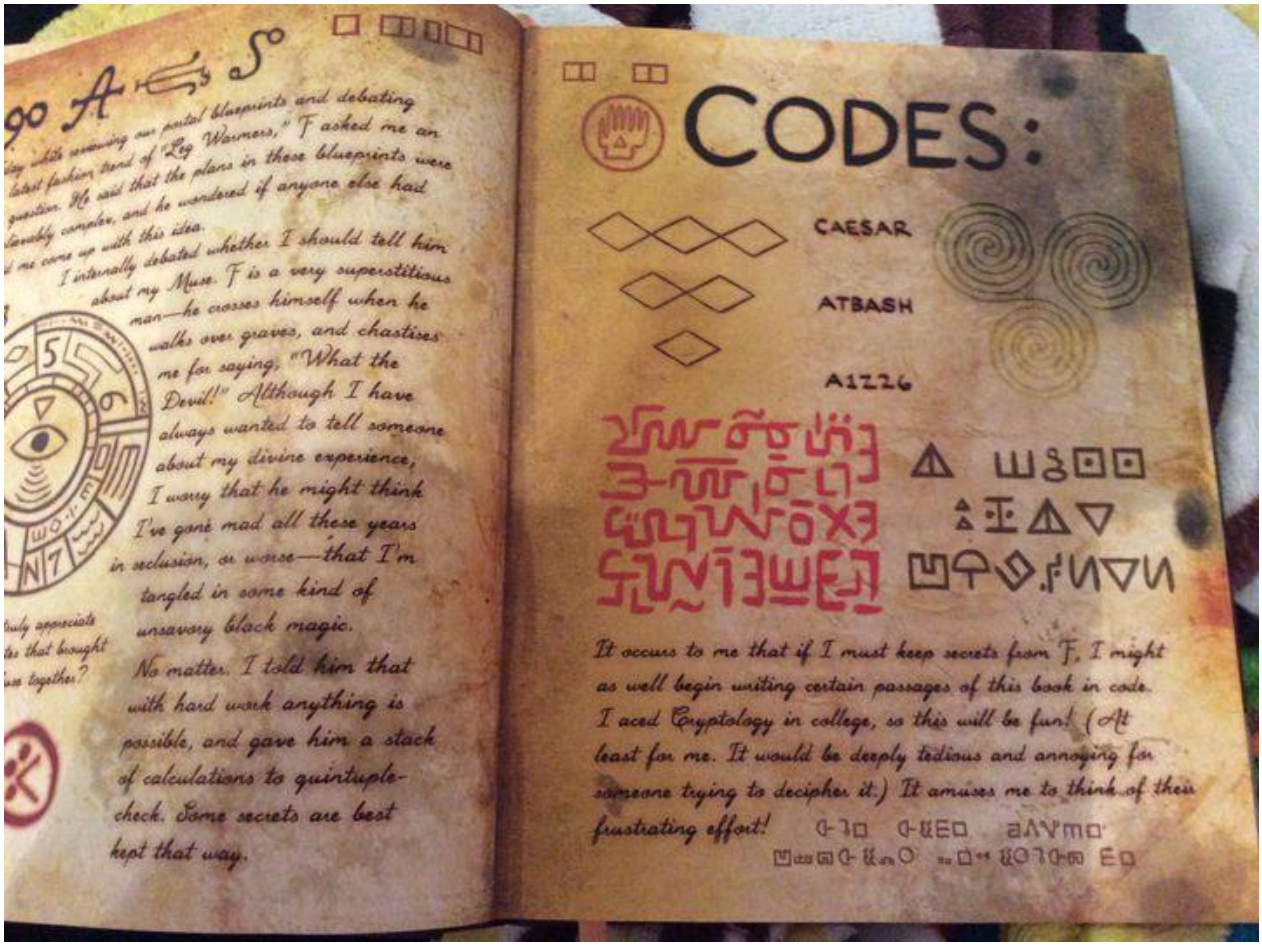
L'argomento **CIFARIO** è un testo che indica la tecnica crittografica utilizzata per cifrare o decifrare; i cifrari attualmente supportati sono: Atbash, Cesare, Sostituzione a sequenza, Parola chiave, Vigenere, Vernam.

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 minuscolo 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

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

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

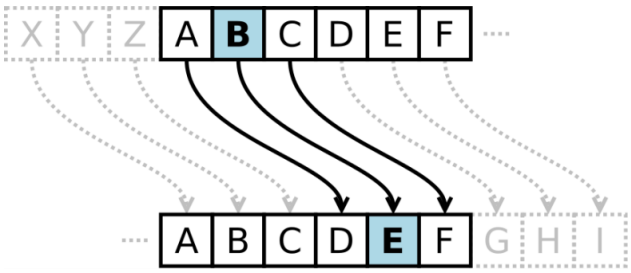
Metodi per violarlo
Una volta scoperto che il messaggio è cifrato con Atbash il cifario è automaticamente violato, poiché esiste una sola chiave.

“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



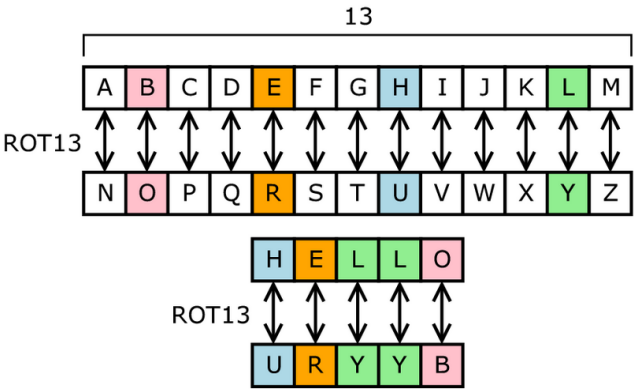
All’epoca di Cesare il cifrario era efficace a causa dell’alto tasso di analfabetismo e ignoranza in materia crittografica.
Il cifrario di Cesare, benché superato da più di mille anni, rimane importante poiché è la **base di vari cifrari più complessi**.

Esempio

```
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 → (ETICHETTA: "messaggio cifrato") (TESTO: messaggio)
```

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

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

```
1  INIZIA
2  messaggio = ottieni testo cifrato → (CIFRARIO: "Sostituzione a sequenza")
3                                      (CHIAVE: "RABSZCTPDWEYOXMUNFQGVHIJKL")
4                                      (TESTO: "CIAO QUESTO MESSAGGIO E' SEGRETO")
5
6  crea casella di testo multilinea → (ETICHETTA: "messaggio cifrato") (TESTO: messaggio)
```

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

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

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

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

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.

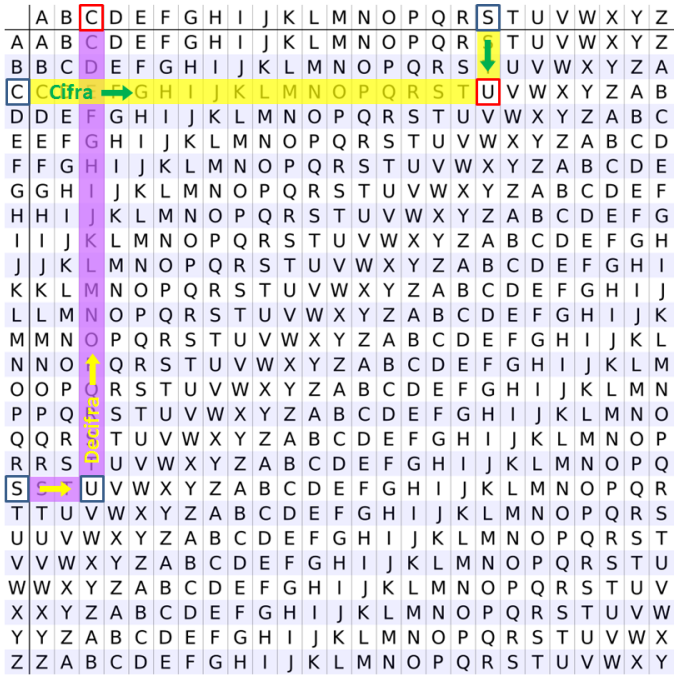
“Vigenere”

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.

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



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)

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	…

Esempio

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

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

Metodi per violarlo
Crittoanalisi, Metodo Kasiski

“Vernam”

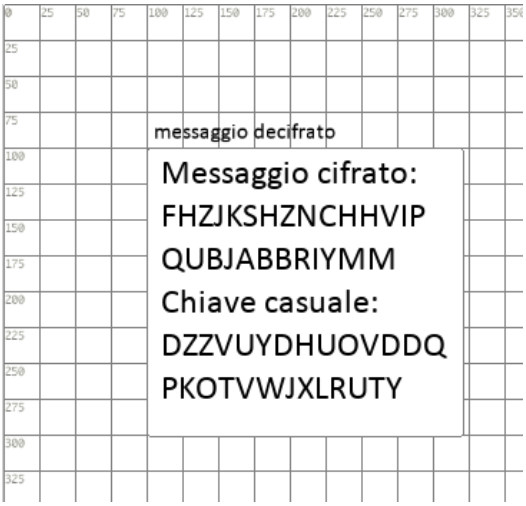
Il cifrario di Vernam è una versione migliorata del cifrario di Vigenère. È l’unico cifrario la cui inviolabilità è stata dimostrata matematicamente, per questo è l’unico cifrario ad essersi guadagno il titolo di “cifrario perfetto”. È stato utilizzato anche durante la guerra fredda per le comunicazioni 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") (TESTO: "CIAO QUESTO MESSAGGIO E' SEGRETO")
3
4  crea casella di testo multilinea → (ETICHETTA: "messaggio cifrato") (TESTO: messaggio)
```



Testo in chiaro: CIAO QUESTO MESSAGGIO E’ SEGRETO
Chiave (generata automaticamente):
DZZVUYDHUOVDDQPKOTVWJXLRUTY
Testo cifrato: FHZJKSHZNCHHVIPQUBJABBRIYMM

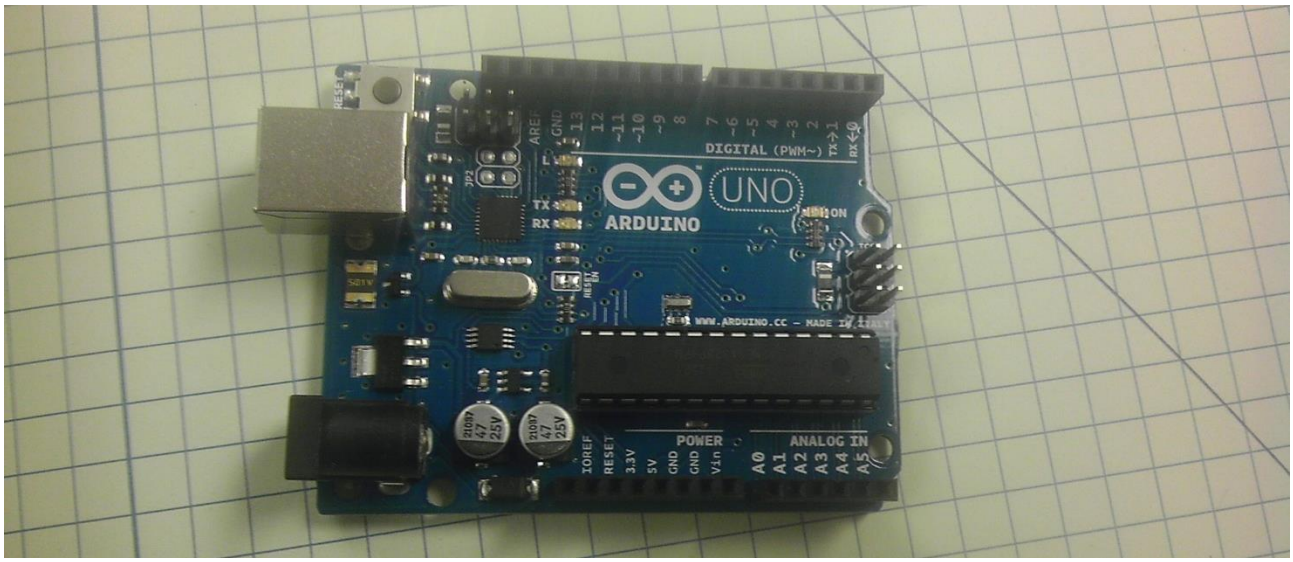
C	I	A	O	Q	U	E	S	T	O	M	E	S	S	A	G	G	I	O	E	S	E	G	R	E	…
D	Z	Z	V	U	Y	D	H	U	O	V	D	D	Q	P	K	O	T	V	W	J	X	L	R	U	…
F	H	Z	J	K	S	H	Z	N	C	H	H	V	I	P	Q	U	B	J	A	B	B	R	I	Y	…

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 e robotica.

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 è specificato viene riconosciuta automaticamente la prima porta a cui è connessa una scheda Arduino
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:

Esempio

```
INIZIA
arduino = ottieni connessione con Arduino

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")}
}

se arduino = "connessione non riuscita"
{
disegna testo → (TESTO: "Arduino non è connesso!")
}
```

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:

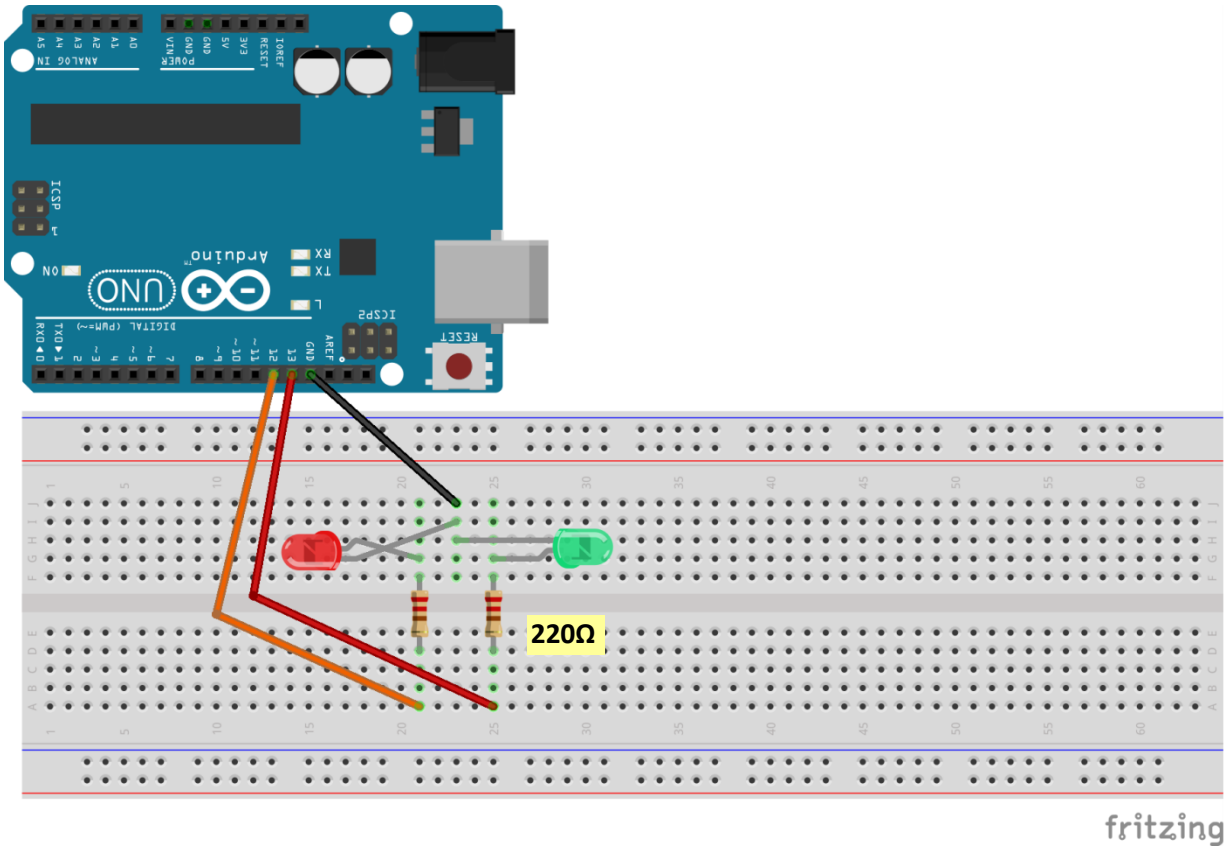
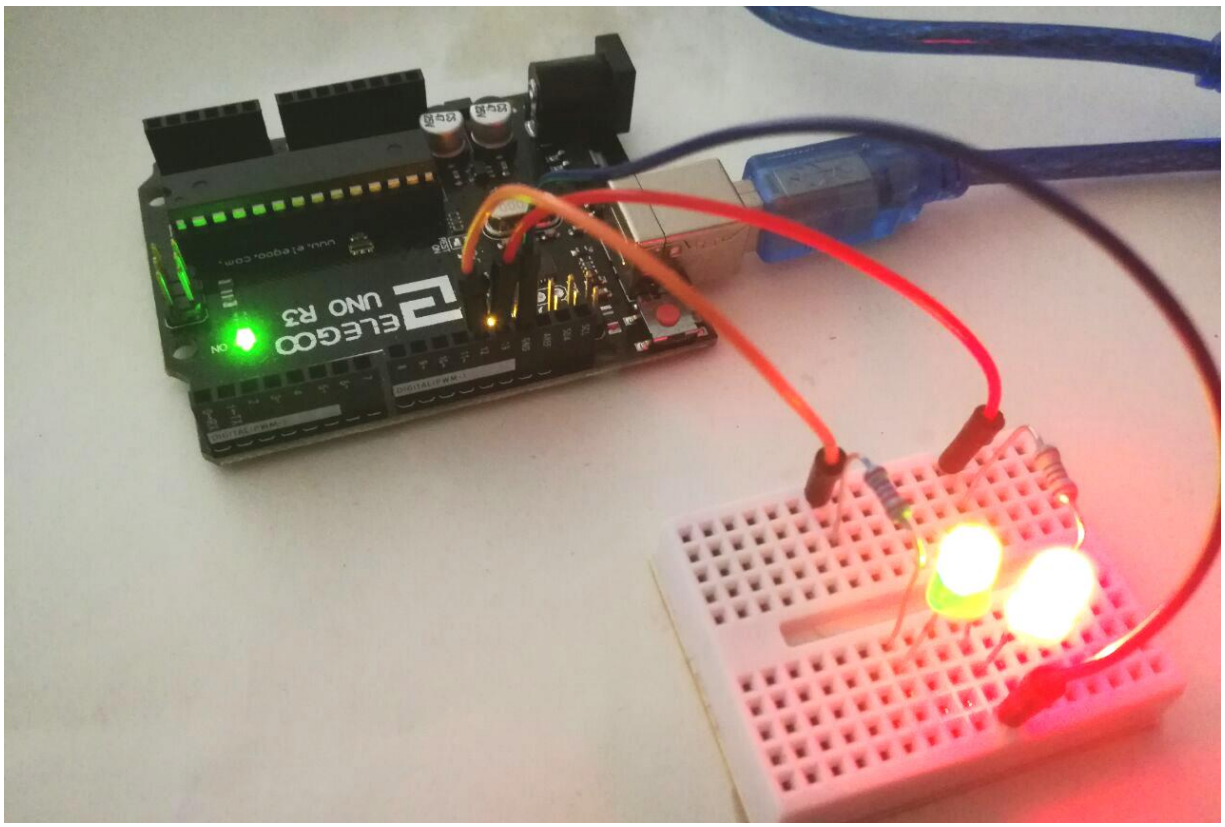


Foto d'esempio:



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.

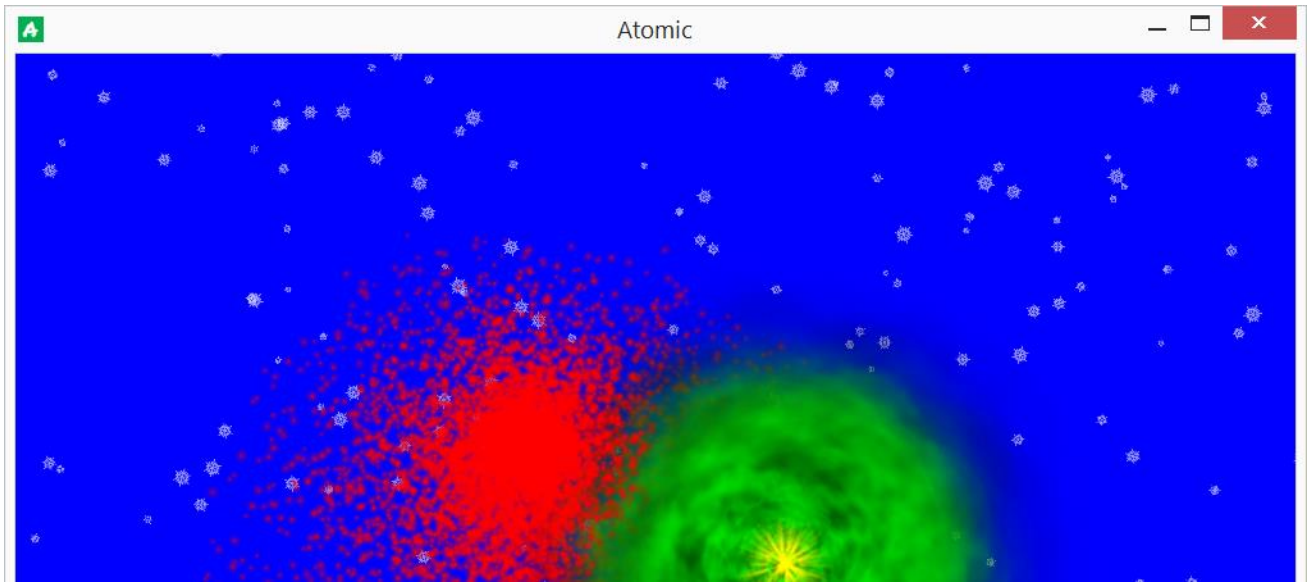
La funzione *Serial.readString()* è molto più lenta rispetto a *Serial.parseInt()*: è sconsigliato eseguire azioni in base alle stringhe ricevute se la temporizzazione è un elemento importante del progetto.

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 sugli effetti grafici particellari



Un sistema particellare è una tecnica grafica che permette di creare effetti visuali animati in modo più veloce rispetto alle tecniche d'animazione tradizionali. Sono utili soprattutto per simulare fenomeni naturali ed altri fenomeni che hanno un aspetto casuale. Alcuni effetti comunemente realizzati con le particelle sono: fuoco, fumo, esplosioni, nuvole, polvere, bagliori, scintille, incantesimi, neve, pioggia…

In Atomic è possibile programmare un gran numero di effetti diversi definendo direttamente le particelle ma esistono anche degli effetti particellari predefiniti di facile utilizzo:

```
disegna effetto speciale animato predefinito → (EFFETTO:) (X:) (Y:) (Z:) (DIMENSIONE:)
```

L'argomento **EFFETTO** definisce l'effetto speciale utilizzato e supporta queste stringhe: "nuvola", "ellisse", "esplosione", "fuoco d'artificio", "bagliore", "pioggia", "anello", "fumo", "fumo che sale", "neve", "scintilla", "stella".

Gli effetti "pioggia" e "neve" non hanno bisogno degli argomenti X, Y e DIMENSIONE.

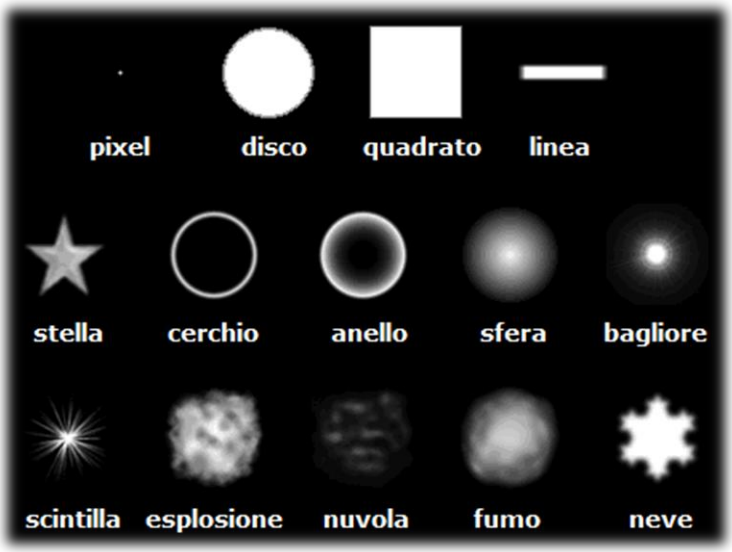
L'argomento **DIMENSIONE** definisce la grandezza dell'effetto e supporta le seguenti stringe: "piccolo", "medio", "grande".

L'argomento **Z** definisce la profondità a cui viene disegnato l'effetto, le stringhe supportate sono due: "sopra" e "sotto".

Per creare e definire una particella esiste una funzione ricca di argomenti: ***ottieni particella***.

```
ottieni particella → (FORMA: ) (DIMENSIONE MINIMA:) (DIMENSIONE MASSIMA:) (SCALA ASSE X:)
                     (SCALA ASSE Y:) (DIREZIONE MINIMA:) (DIREZIONE MASSIMA:)
                     (ROTAZIONE MINIMA:) (ROTAZIONE MASSIMA:) (VELOCITA MINIMA:)
                     (VELOCITA MASSIMA:) (DURATA MASSIMA) (DURATA MINIMA:) (COLORE INIZIALE:)
                     (COLORE MEDIANO:) (COLORE FINALE:) (TRASPARENZA INIZIALE)
                     (TRASPARENZA MEDIANA:) (TRASPARENZA FINALE:) (CRESCITA:) (ACCELERAZIONE:)
                     (VARIAZIONE VELOCITA:) (VARIAZIONE DIREZIONE:) (VARIAZIONE ROTAZIONE:)
                     (INCREMENTO ROTAZIONE:) (INCREMENTO DIREZIONE:) (COLORE ADDITIVO: )
```

L’argomento **FORMA** definisce l’aspetto della particella e supporta le seguenti stringhe: "pixel", "nuvola", "cerchio", "disco", "esplosione”, "bagliore", ”linea”, "anello", "fumo", ”neve", "scintilla", "sfera", "quadrato", "stella".



Gli argomenti **DIMENSIONE MASSIMA** e **DIMENSIONE MINIMA** servono per definire in modo relativo la grandezza delle particelle: ad esempio se impostati su 1 e 10 verranno create casualmente particelle grandi fino ad un massimo di 10 volte la dimensione minima.

Gli argomenti **SCALA ASSE X** e **SCALA ASSE Y** definiscono le dimensioni assolute della particella rispetto alla forma originale.

Gli argomenti **DIREZIONE MINIMA** e **DIREZIONE MASSIMA** definiscono i limiti della direzione di una particella quando viene creata.

Gli argomenti **ROTAZIONE MINIMA** e **ROTAZIONE MASSIMA** definiscono i limiti della rotazione di una particella alla creazione.

Gli argomenti **VELOCITA MINIMA** e **VELOCITA MASSIMA** definiscono i limiti della velocità di una particella alla creazione.

Gli argomenti **DURATA MINIMA** e **DURATA MASSIMA** definiscono i limiti del tempo d’esistenza di una particella (in 1/30 secondi).

Gli argomenti **COLORE INIZIALE**, **COLORE MEDIANO** e **COLORE FINALE** indicano il colore della particella durante il tempo della sua esistenza. Il colore sfuma da un valore all’altro.

Gli argomenti **TRASPARENZA INIZIALE**, **TRASPARENZA MEDIANA** e **TRASPARENZA FINALE** indicano l’opacità della particella durante il tempo della sua esistenza. La trasparenza sfuma da un valore all’altro.

L’argomento **CRESCITA** definisce l’ingrandimento nel tempo della particella in base alla scala.

L’argomento **ACCELERAZIONE** definisce l’aumento di velocità della particella nel tempo.

L’argomento **INCREMENTO DIREZIONE** definisce l’aumento nel tempo della direzione della particella.

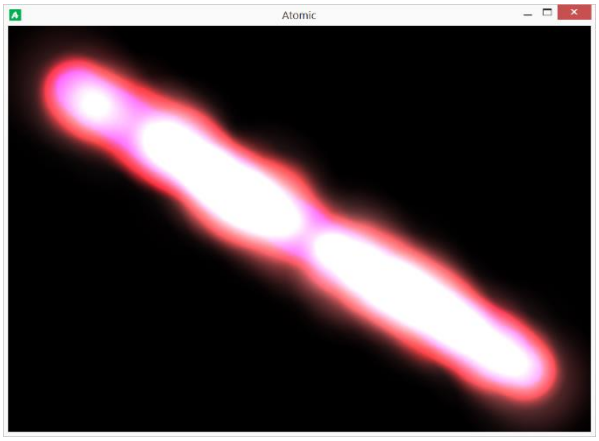
L’argomento **INCREMENTO ROTAZIONE** definisce l’auemnto nel tempo della rotazione della particella.

L’argomento **VARIAZIONE VELOCITA** definisce una variazione casuale nel tempo della velocità della particella.

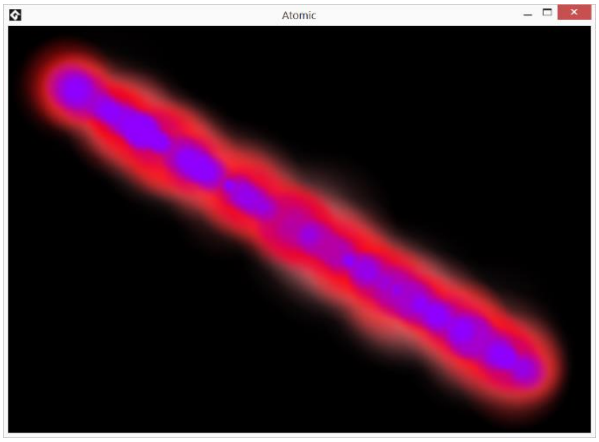
L’argomento **VARIAZIONE DIREZIONE** definisce una variazione casuale nel tempo della direzione della particella.

L’argomento **VARIAZIONE ROTAZIONE** definisce una variazione casuale nel tempo della rotazione della particella.

L'argomento **COLORE ADDITIVO** imposta la sintesi del colore utilizzata (vero = additiva, falso = sottrattiva). La sintesi additiva somma la luminosità dei colori: questo genera un effetto visivo luminescente.



(COLORE ADDITIVO: vero)



(COLORE ADDITIVO: falso)

N.B: se si utilizza la sintesi additiva le particelle saranno invisibili su sfondo bianco e difficilmente visibili su uno sfondo chiaro. Per ottenere un buon effetto di luminescenza bisogna utilizzare un colore di sfondo scuro.

Per disegnare le particelle ci sono due metodi: disegnarla direttamente o usare un generatore di particelle.

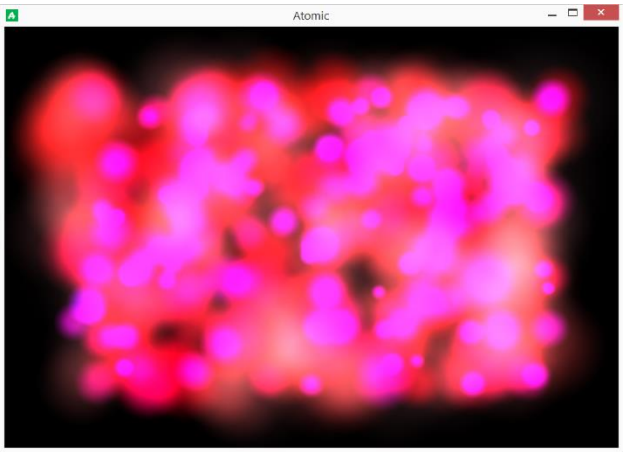
Per disegnare delle particelle la funzione più semplice è disegna *direttamente delle particelle*:

```
disegna direttamente delle particelle → (NOME:) (X:) (Y:) (NUMERO:)
```

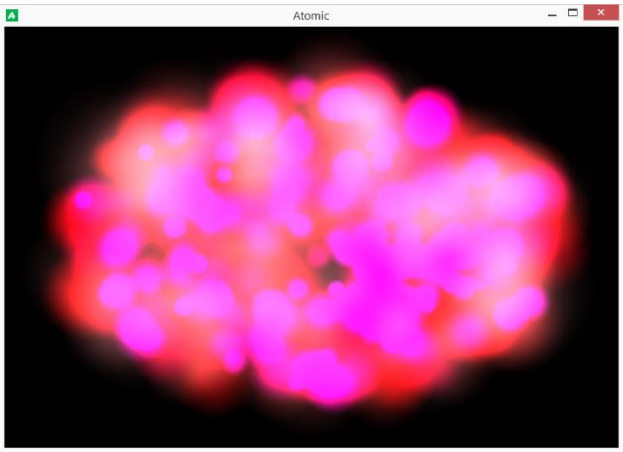
Utilizzare un generatore di particelle significa definire un'area che emette automaticamente delle particelle nelle modalità specificate. Per creare un generatore la funzione da utilizzare è *ottieni generatore di particelle*:

```
ottieni generatore di particelle → (X:) (Y:) (ALTEZZA:) (LARGHEZZA:) (FORMA:) (DISTRIBUZIONE:)
```

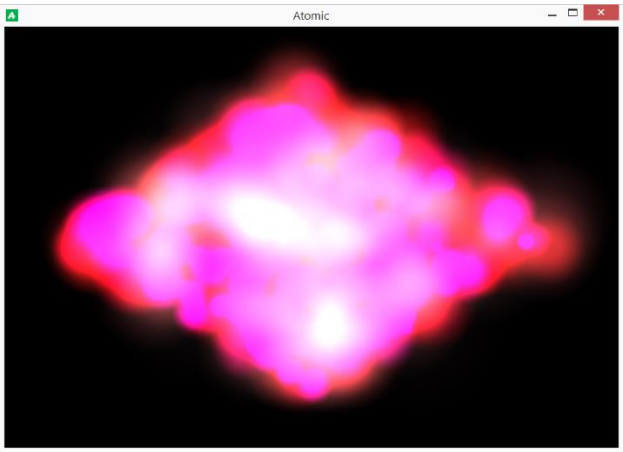
L’argomento **FORMA** definisce la forma dell’area, le stringhe utilizzabili sono: “rettangolo”, “ellisse”, “rombo”, “linea”.



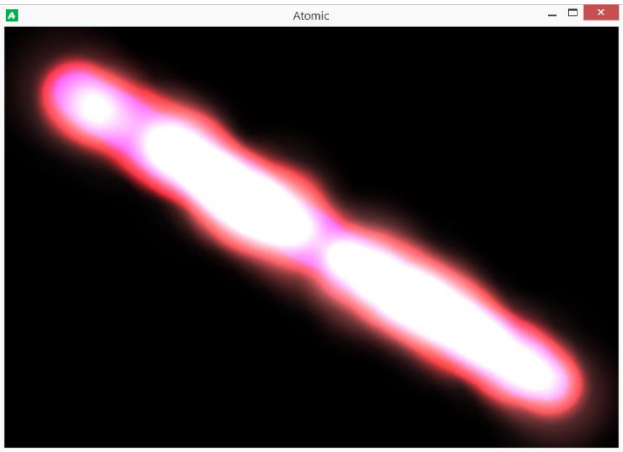
(FORMA: “rettangolo”)



(FORMA: “ellisse”)



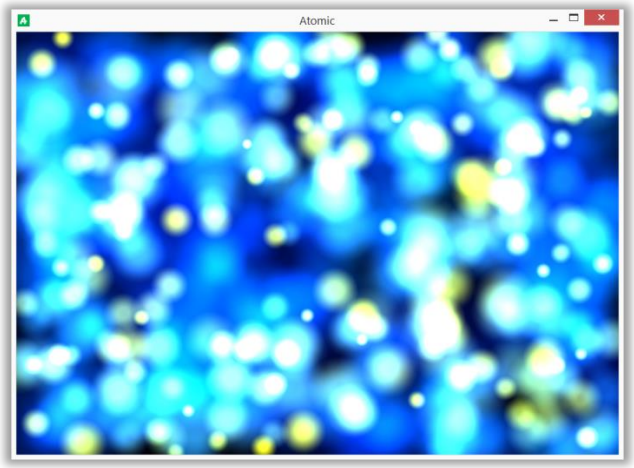
(FORMA: “rombo”)



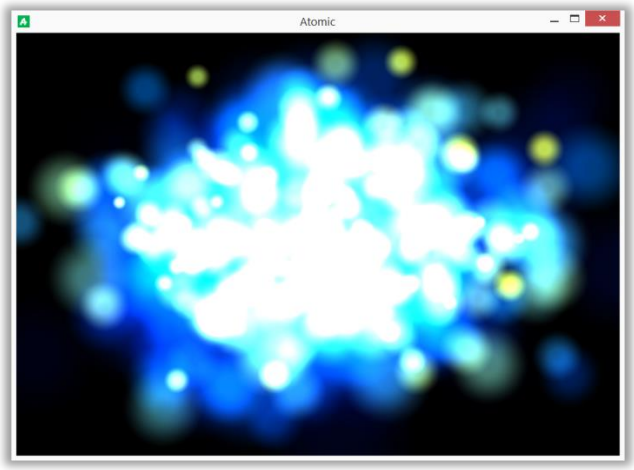
(FORMA: “linea”)

L'argomento **DISTRIBUZIONE** definisce il modo in cui viene distribuita la generazione delle particelle sulla superficie.
Le stringhe utilizzabili sono: "omogenea", "verso il centro", "verso il bordo".

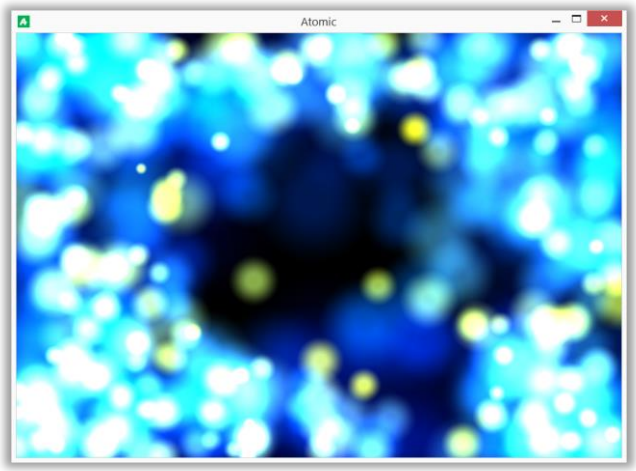
La distribuzione **"omogenea"** distribuisce la generazione delle particelle in modo lineare:
le particelle generate tenderanno ad occupare in modo uniforme l'area.



La distribuzione **"verso il centro"** è di tipo gaussiana:
le particelle generate tenderanno ad accumularsi verso il centro.



La distribuzione **"verso il bordo"** è di tipo gaussiana inversa:
le particelle generate tenderanno ad accumularsi verso il bordo esterno dell'area.



Una volta ottenuto un generatore di particelle ci sono due modi per utilizzarlo: attivandolo in modo perpetuo una volta sola all'inizio o attivandolo a comando solo per un attimo:

```
attiva generatore di particelle → (NOME:) (PARTICELLA:) (NUMERO:)
```

```
attiva generatore di particelle solo per un istante → (NOME:) (PARTICELLA:) (NUMERO:)
```

L'argomento **NUMERO** indica il numero di particelle da generare ogni 1/30 secondi.

Esempio completo

```
INIZIA
generatore = ottieni generatore di particelle → (X: 100) (Y: 100) (LARGHEZZA: larghezza finestra-100)
                                                (ALTEZZA: altezza finestra-100) (FORMA: "ellisse")
                                                (DISTRIBUZIONE: "omogenea")

particella = ottieni particella → (FORMA: "sfera") (COLORE INIZIALE: viola) (COLORE MEDIANO: rosso)
                                  (COLORE FINALE: rosa) (DIREZIONE MINIMA: 0) (DIREZIONE MASSIMA: 360)
                                  (VELOCITA MINIMA: 0) (VELOCITA MASSIMA:0) (DIMENSIONE MINIMA: 1)
                                  (DIMENSIONE MASSIMA: 5) (SCALA ASSE X: 0.25) (SCALA ASSE Y: 0.25)

attiva generatore di particelle → (NOME: generatore) (PARTICELLA: particella) (NUMERO: 5)
colore sfondo = nero

CICLO CONTINUO
se tasto destro del mouse è premuto = vero
{
disegna direttamente delle particelle → (NOME: particella) (NUMERO: 5)
                                        (X: x del mouse) (Y: y del mouse)
}

se tasto sinistro del mouse è premuto = vero
{
attiva generatore di particelle solo per un istante → (NOME: generatore) (PARTICELLA: particella)
                                                      (NUMERO: 15)
}
```

È possibile disegnare effetti particellari predefiniti nella posizione di determinati esemplari di oggetti in particolari situazioni:

```
disegna effetto speciale su un elemento → (ELEMENTO:) (EFFETTO:) (DIMENSIONE:)
                                          (DIMENSIONE:) (COLORE:) (Z:)
```

Disegna l'effetto quando viene chiamata la funzione .

```
disegna effetto speciale su un elemento quando viene creato → (ELEMENTO:) (EFFETTO:)
                                                               (DIMENSIONE:) (COLORE:) (Z:)
```

Disegna l'effetto quando l'elemento specificato viene creato.

```
disegna effetto speciale su un elemento quando viene distrutto → (ELEMENTO:) (EFFETTO:)
                                                                  (DIMENSIONE:) (COLORE:) (Z:)
```

Disegna l'effetto quando l'elemento specificato viene distrutto.

L'argomento **EFFETTO** definisce l'effetto speciale utilizzato e supporta queste stringhe: "nuvola", "ellisse", "esplosione", "fuoco d'artificio", "bagliore", "pioggia", "anello", "fumo", "fumo che sale", "neve", "scintilla", "stella".

L'argomento **DIMENSIONE** definisce la grandezza dell'effetto e supporta le seguenti stringe: "piccolo", "medio", "grande".

L'argomento **Z** definisce la profondità a cui viene disegnato l'effetto, le stringhe supportate sono due: "sopra" e "sotto".

Esempio

```
1  asteroide_distrutto = ottieni risultato controllo collisione tra → (ELEMENTO 1: proiettile)
2                                                                     (ELEMENTO 2: asteroide)
3
4  se asteroide_distrutto = vero { suona → (SUONO: "suoni/esplosione.ogg") aumenta punteggio di 1 }
5
6  disegna effetto speciale su un elemento quando viene distrutto → (EFFETTO: "esplosione")
7                                                                  (DIMENSIONE: "grande")
8                                                                  (ELEMENTO: asteroide)
9                                                                  (COLORE: grigio)
10
11 distruggi questi elementi quando collidono tra loro → (ELEMENTO 1: proiettile) (ELEMENTO 2: asteroide)
```

Funzioni miscellanea

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

```
imposta griglia → (VISIBILE:) (COLORE SFONDO:) (COLORE LINEE: ) (DIMENSIONE:)
```

Imposta la dimensione e la grafica dei quadretti della griglia di sfondo della finestra. N.B. impostando un colore di sfondo tramite la variabile integrata “colore sfondo”, la griglia si disattiva automaticamente.

```
salva schermata
```

Salva la schermata attuale in un file immagine .png; quando la funzione viene chiamata si apre una schermata “Salva con nome” per salvare il file con il nome desiderato, nella cartella desiderata.

```
crea gif animata → (X:) (Y:) (LARGHEZZA:) (ALTEZZA:) (SECONDI:) (NOME:)
```

Salva le schermate nel tempo creando un file d’immagine animata .gif.

Esempio

```
1 se tasto invio è stato premuto = vero
2 {
3 crea gif animata → (X: 0) (Y: 0) (LARGHEZZA: larghezza finestra) (ALTEZZA: altezza finestra)
4                    (NOME: "immagini/test.gif") (SECONDI: 3)
5 }
```

Una volta lanciata la funzione gli unici argomenti modificabili sono X e Y. In questo modo è possibile modificare l’animazione visualizzata come se fosse ripresa da una telecamera.

Esempio

```
1 se tasto invio è stato premuto = vero
2 {
3 crea gif animata → (X: x del mouse) (Y: y del mouse) (LARGHEZZA: 300) (ALTEZZA: 300)
4                    (NOME: "immagini/test.gif") (SECONDI: 3)
5 }
```

```
esci dal programma
```

Interrompe l’esecuzione del programma, tornando all’editor.

```
riavvia il programma
```

Riavvia il programma, reinterpretando il codice.

```
imposta schermo intero
```

Imposta la finestra a schermo intero, nascondendo anche i bordi e le icone in alto. N.B: In base alla dimensione e risoluzione del monitor utilizzato e alle dimensioni assegnate alla finestra il contenuto visualizzato potrebbe risultare deformato.

AUMENTA E DIMINUISCI

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

Per incrementare è possibile utilizzare questa forma:

```
aumenta  <variabile> di <valore>
```

oppure, per diminuire:

```
diminuisci  <variabile> di <valore>
```

Esempio

```
1  aumenta orsetto_lavatore di 1
```

Esempio

```
1  diminuisci orsetto_lavatore di 1
```

Si può ottenere il risultato equivalente del costrutto aumenta scrivendo l'operazione in forma estesa:

```
<variabile x> = <variabile x> + <valore>
```

Ovvero somma un valore al valore di se stesso

In altri linguaggi l'operazione d'incremento può essere scritta in questo modo:

```
orsetto_lavatore+=1;
```

o addirittura

```
orsetto_lavatore++;
```

Queste forme (non supportate in Atomic) sono sicuramente veloci e comode per un programmatore navigato ma molto distanti dal linguaggio umano.

è anche possibile diminuire o aumentare in base al risultato di una espressione.

Esempio

```
1  diminuisci orsetto_lavatore di 5*2
2  aumenta criceto di gatto
3  diminuisci topo_ragno di 1+2*(100-6/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 "//".

```
//testo del commento
```

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) (COLORE: verde)
3 //disegna cerchio --> (RAGGIO: 100) (X: 500) (Y: 600) (COLORE: blu)
4 disegna cerchio → (RAGGIO: 50) (X: 100) (Y: 150) //(COLORE: rosso)
```

La prima e la terza linea verranno ignorate dal computer.
L'ultima parte della quarta linea verrà ignorata: il cerchio non verrà colorato di rosso.

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 "*/".

```
/*testo
del
commento*/
```

Esempio

```
1 /*Tutto questo pezzo di codice verrà ignorato (non produrrà nessun risultato)
2 disegna cerchio --> (RAGGIO: 200) (X: 300) (Y: 400) (COLORE: verde)
3 disegna cerchio --> (RAGGIO: 100) (X: 500) (Y: 600) (COLORE: blu)
4 disegna cerchio --> (RAGGIO: 50) (X: 100) (Y: 150) (COLORE: rosso)*/
```

COSTANTI

Le costanti sono valori non modificabili già integrati in Atomic.
Immutabilità a parte, possono essere utilizzate nello stesso modo delle variabili.

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

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”
è uguale a	“=“
è maggiore di	“>“
è minore di	“<“
è maggiore o uguale a	“>=“
è minore o uguale a	“<=“
è diverso da	“!=“

N.B. non utilizzabili nelle espressioni!

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

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
blu ardesia		112,152,192
bronzo		205,127,50
castagno		205,92,92
castagno scuro		152,105,96
ciano		0,255,255
corallo		255,127,80
giada		0,168,107
giallo		255,255,0
grigio		192,192,192
grigio chiaro		242,242,242
grigio scuro		129,128,128
magenta		255,0,255
malva		224,176,255
marrone		128,0,0
nero		0,0,0
oro		255,192,0
rosa		255,192,203
rosa vivo		255,0,127
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
giada		0,168,107
viola		143,0,255

Costanti che rappresentano note 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

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 logica è vera>   allora   <esegui questo codice>   .
```

L'utilizzo più semplice e comune è la forma:

```
se   <variabile>   <confronto>   <valore>   allora   <esegui questo codice>   .
```

Esempio

```
1  se gatto = 1 allora disegna cerchio → (RAGGIO: 100) (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 di confronto:

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	≠

Esempio

```
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)
5                                                            (COLORE: arancione) .
```

È possibile utilizzare la forma estesa (descrizione, vedi costanti) **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)
5                                                                             (COLORE: arancione) .
```

È anche possibile inserire più righe di codice controllate da un *se*.

Esempio

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

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

```
se   <espressione>   <confronto>   <espressione>
{
 <esegui questa linea di codice>
 <esegui questa linea di codice>
 <esegui questa linea di codice>
}
```

Parentesi graffa aperta: { = allora
Parentesi graffa chiusa: } = .

Esempio

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

Nulla vieta di usare questa notazione anche per delle singole linee.

```
se   <espressione>   <confronto>   <espressione>   {   <esegui questo codice>  }
```

Esempio

```
se gatto = 1 { disegna cerchio → (RAGGIO: 100) (COLORE: blu) }
```

È 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:

Esempio

```
INIZIA
gatto = 0
cane = 1

CICLO CONTINUO
se gatto = 0
{
    se cane < 101
    {
    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 ma solo se la variabile gatto equivale a zero.

È altresì possibile fare confronti di uguaglianza con stringhe di testo.

Esempio

```
INIZIA
gatto = "persiano"

CICLO CONTINUO
se gatto = "persiano" { disegna cerchio → (RAGGIO: 100) }
```

N.B. Se si confrontano stringhe di testo gli unici operatori di confronto utilizzabili sono “=” e “!=” e la loro relativa forma estesa “è uguale a” e “è diverso da”.

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 beep2)
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

```
INIZIA
tempo = 5*30

CICLO CONTINUO
diminuisci tempo di 1

se tempo = 0
{
suona → (SUONO: suono beep2)
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** o **espressioni logiche**.

Operatori logici unari

L'unico operatore unario supportato è la negazione "non".

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

Semplicemente la negazione inverte il vero con il falso e viceversa.
L'operatore **non**, va utilizzato scrivendolo prima del valore che si vuole negare.

```
x = non y
```

Esempio

```
1  cane = vero
2  gatto = non cane
```

Il valore di gatto sarà 0 (falso).

È anche possibile invertire il valore della variabile negando la variabile stessa.

Esempio

```
1  INIZIA
2  luce_accesa = vero
3
4  CICLO CONTINUO
5  se tasto invio è stato premuto = vero { luce_accesa = non luce_accesa }
6  se luce_accesa = vero { colore sfondo = giallo }
7  se luce_accesa = falso { colore sfondo = nero }
```

Questo esempio spegne/accende una luce premendo il tasto invio.

Questa logica è riassumibile in 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	||	v
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>   .
```

Esempio

```
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 v y)=1 solo se (x=0 ∧ y=1) v (x=1 ∧ y=0) v (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 seguendo l'ordine di priorità: non, e, o, o esclusivamente.
Come per le espressioni aritmetiche **per modificare l'ordine di priorità dei calcoli è possibile utilizzare le parentesi tonde**.

Esempio

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

In una espressione logica si può evitare di scrivere il confronto con il vero ("=1", "=vero"), poiché, se non viene confrontata con un altro valore, una variabile restituirà il suo valore. Un utilizzo tipico è il controllo dello stato dei tasti della tastiera.

Esempio

```
1  se tasto invio è premuto e tasto A è premuto { disegna cerchio }
```

COSTRUTTO RIPETI FINCHE

Il costrutto ***ripeti finche*** (leggi finché, l'accento corretto è omesso per facilitare la scrittura) ha una sintassi simile al costrutto *se*:

```
ripeti finche   <espressione logica>>    {   <esegui questo codice>   }
```

Esempio

```
1  ripeti finche gatto < 100 { aumenta gatto di 1 }
```

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

Esempio

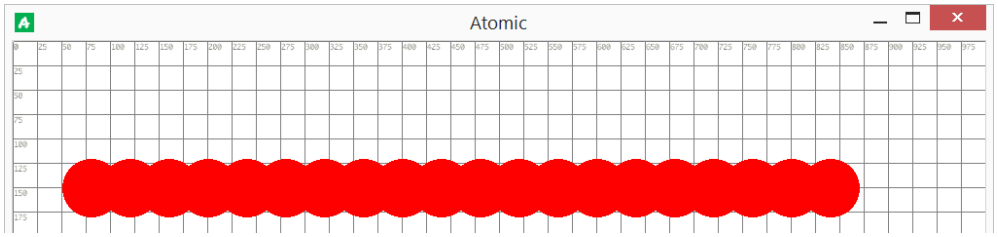
```
1  gatto = 100
```

Perché allora utilizzare un ciclo *ripeti finche*?
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. Mettiamo il caso che vogliamo disegnare 20 cerchi equidistanti; possiamo scrivere:

Esempio

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

Questo codice è corretto ma è lungo e ridondante.



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

Esempio

```
1 cerchi = 1
2 ripeti finche cerchi < 20
3 {
4 disegna cerchio → (RAGGIO: 30) (COLORE: rosso) (X: 40*cerchi)
5 aumenta cerchi di 1
6 }
```

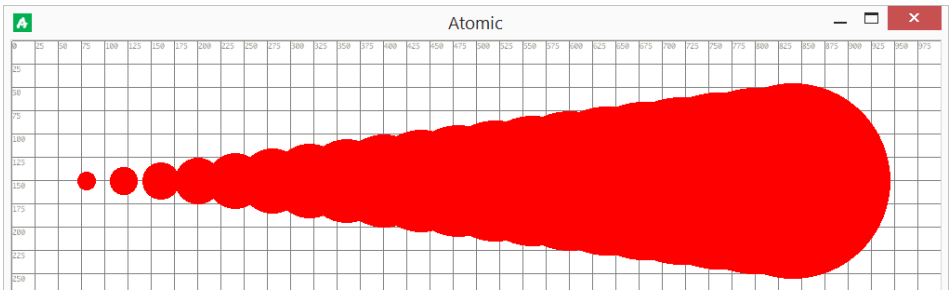
Il funzionamento è il seguente:

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

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

Esempio

```
1 cerchi = 1
2 ripeti finche cerchi < 20
3 {
4 disegna cerchio → (RAGGIO: 5*cerchi) (COLORE: rosso) (X: 40*cerchi)
5 aumenta cerchi di 1
6 }
```



Gli operatori logici **e**, **o** ed **o esclusivamente** sono applicabili anche ai cicli *ripeti finche*.
All'interno di un ciclo *ripeti finche* possono essere presenti anche altre istruzioni condizionali.
Il costrutto *ripeti finche* è lo strumento più potente presente in Atomic, le sue applicazioni sono infinite e permettono di automatizzare una miriade di compiti ripetitivi.

ATTENZIONE A NON CREARE CICLI INFINITI!

I cicli *ripeti finche* eseguono il codice finché 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:

Esempio

```
1 prova = 150
2 ripeti finche prova > 100
3 {
4 disegna cerchio → (RAGGIO: 30) (COLORE: rosso) (X: 40*prova)
5 aumenta prova di 1
6 }
```

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 *ripeti finche* e in molti casi può sostituirlo.

La sintassi è la seguente:

```
ripeti per   <espressione>   volte   :   <esegui questo codice>   .
```

Esempio

```
1  cerchi = 1
2  ripeti per 20 volte: disegna cerchio → (RAGGIO: 30) (COLORE: rosso) (X: 40*cerchi)
3  aumenta cerchi di 1 .
```

La differenza con il ciclo *ripeti 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 *ripeti 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  cerchi = 1
2  ripeti per 20 volte
3  {
4  disegna cerchio → (RAGGIO: 30) (COLORE: rosso) (X: 40*cerchi)
5  aumenta cerchi di 1
6  }
```

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 nome può anche essere una stringa di testo, ad esempio "Tabella libri preferiti".

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 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: "325|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:)
```

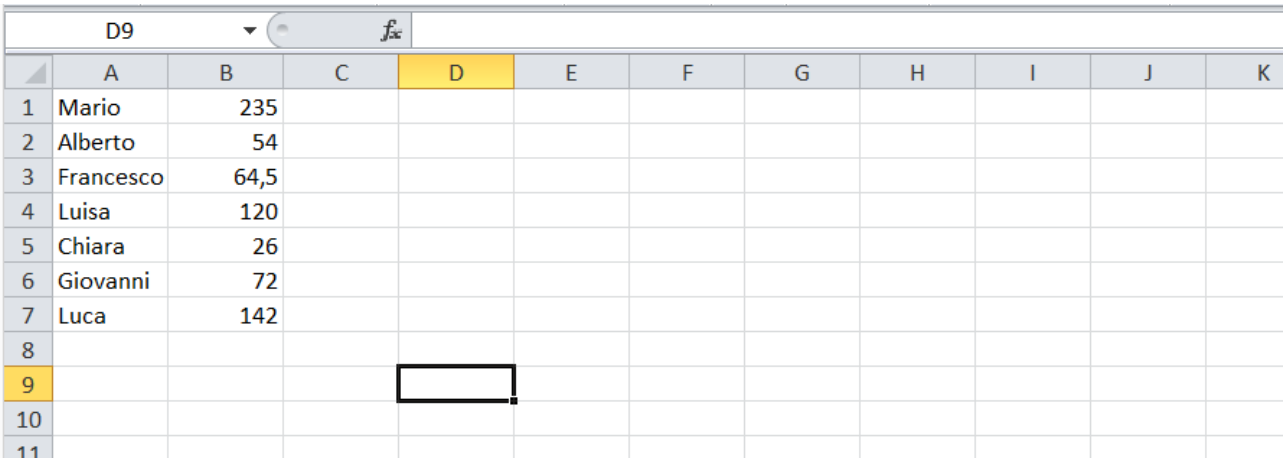
è possibile esportare tabelle in formato . csv (compatibili con Microsoft Excel e altri programmi simili).

Esempio

```
1  INIZIA
2  crea tabella → (NOME: "Esempio tabella")
3  riempi tabella → (NOME: "Esempio tabella") (COLONNA 1: "Mario|Alberto|Francesco|Luisa|Chiara|Giovanni|Luca|")
4                                             (COLONNA 2: "235|54|64.5|120|26|72|142|")
5
6  CICLO CONTINUO
7  disegna tabella → (NOME: "Esempio tabella") (RIGHE: 7) (COLONNE: 2)
8
9  se tasto invio è stato premuto = vero { esporta tabella → (NOME: "Esempio tabella") (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 (e altri programmi simili).



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											

Importare una tabella

```
importa tabella → (NOME:) (INDIRIZZO:)
```

Analogamente all'esportazione è possibile importare all'interno di Atomic tabelle in formato .csv.
In base al nome specificato la funzione crea una nuova tabella o sovrascrive una tabella esistente.
Il file deve trovarsi all'interno di *%LOCALAPPDATA%/Atomic* o di una sua sottocartella.

Esempio

```
1  INIZIA
2  importa tabella → (NOME: esempio) (INDIRIZZO: "esempio_tabella.csv")
3
4  CICLO CONTINUO
5  disegna tabella → (NOME: esempio) (RIGHE: 7) (COLONNE: 2)
```

Funzioni avanzate sulle tabelle: **! [SPERIMENTALI]** Queste funzioni non sono ancora state implementate nella versione ufficiale.

```
ottieni il valore più basso nell'area indicata di questa tabella → (NOME:) (X:) (Y:) (BASE:) (ALTEZZA:)
ottieni il valore più alto nell'area indicata di questa tabella → (NOME:) (X:) (Y:) (BASE:) (ALTEZZA:)
ottieni il valore più vicino alla media nell'area indicata di questa tabella → (NOME:) (X:) (Y:)
                                                                               (BASE:) (ALTEZZA:)
ottieni il valore medio nell'area indicata di questa tabella → (NOME:) (X:) (Y:) (BASE:) (ALTEZZA:)
ottieni la sommatoria dell'area indicata di questa tabella → (NOME:) (X:) (Y:) (BASE:) (ALTEZZA:)
ottieni risultato ricerca in questa tabella → (NOME:) (X:) (Y:) (BASE:) (ALTEZZA:) (RICERCA:)
                                              (OTTIENI: riga, colonna, occorrenze, cella a sinistra,
                                               cella a destra, cella sopra, cella sotto)
mischia a caso le celle di questa tabella nell'area indicata → (NOME:) (X:) (Y:) (BASE:) (ALTEZZA:)
ordina in ordine crescente 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 crescente 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:)
copia l'area indicata di questa tabella in un'altra tabella → (NOME:) (X:) (Y:) (BASE:) (ALTEZZA:)
                                                             (DESTINAZIONE:) (X DESTINAZIONE:) (Y: DESTINAZIONE:)
elimina riga di questa tabella → (NOME:) (RIGA:)
elimina colonna di questa tabella → (NOME:) (RIGA:)
cancella i dati nell'area indicata di questa tabella → (NOME:) (X:) (Y:) (BASE:) (ALTEZZA:)
cancella le lettere nell'area indicata di questa tabella → (NOME:) (X:) (Y:) (BASE:) (ALTEZZA:)
rendi il testo tutto in maiuscolo nell'area indicata di questa tabella → (NOME:) (X:) (Y:) (BASE:) (ALTEZZA:)
aggiungi testo nell'area indicata di questa tabella → (NOME:) (X:) (Y:) (BASE:) (ALTEZZA:)
                                                      (POSIZIONE: "prima"|"dopo"|numero) (TESTO:)
sostituisci valore nell'area indicata di questa tabella → (NOME:) (X:) (Y:) (BASE:) (ALTEZZA:)
                                                          (VALORE:) (NUOVO VALORE:) (SOSTITUISCI PAROLA INTERA:)
imposta formattazione condizionale nell'area indicata di questa tabella → (NOME:) (X:) (Y:) (BASE:) (ALTEZZA:)
                                                                          (CONDIZIONE: es. VALORE>100)
                                                                          (COLORE TESTO:) (COLORE SFONDO:)
```

DEFINIRE LE PROPRIE FUNZIONI

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

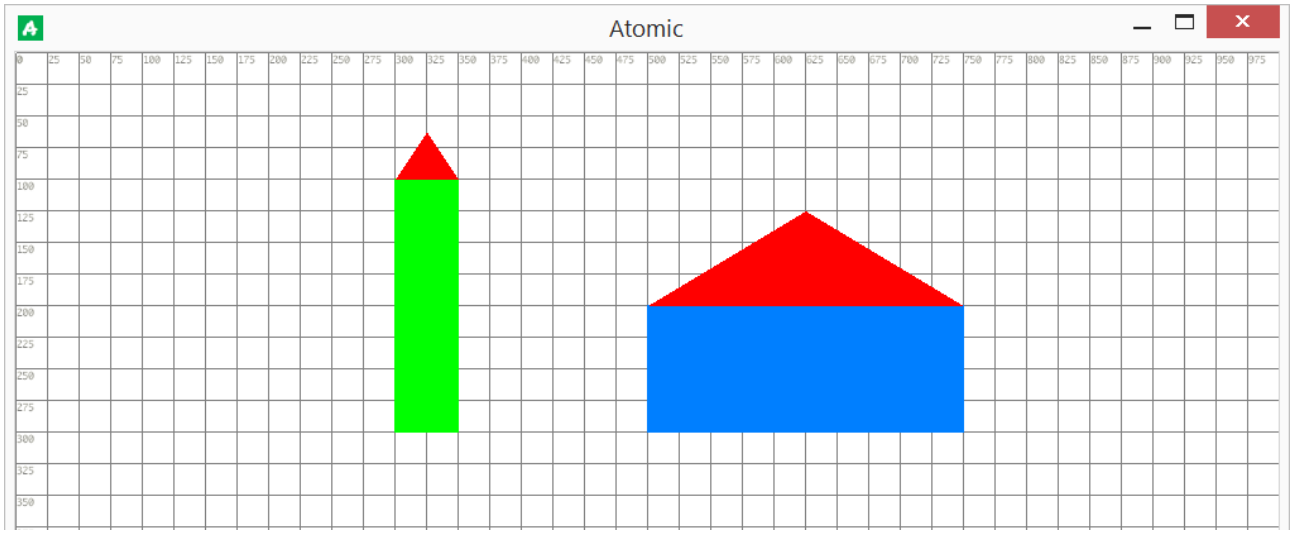
```
definisci funzione “nome funzione”
{
… codice …
}
```

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 ripeti *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

```
1  definisci funzione "disegna casa"
2  {
3  disegna rettangolo → (X: <"X">) (Y: <"Y">-<"ALTEZZA">) (BASE: <"LARGHEZZA">)
4                       (ALTEZZA: <"ALTEZZA">) (COLORE: <"COLORE">)
5  disegna triangolo → (X 1: <"X">) (Y 1: <"Y">-<"ALTEZZA">) (X 2: <"X">+<"LARGHEZZA">)
6                      (Y 2: <"Y">-<"ALTEZZA">) (X 3:  <"X">+<"LARGHEZZA">/2)
7                      (Y 3: <"Y">-<"ALTEZZA">-100/[<"ALTEZZA">/75]) (COLORE: rosso)
8  }
9
10 disegna casa → (LARGHEZZA: 250) (ALTEZZA: 100) (X: 500) (Y: 300) (COLORE: azzurro)
11 disegna casa → (LARGHEZZA: 50) (ALTEZZA: 200) (X: 300) (Y: 300) (COLORE: verde)
12
```



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:

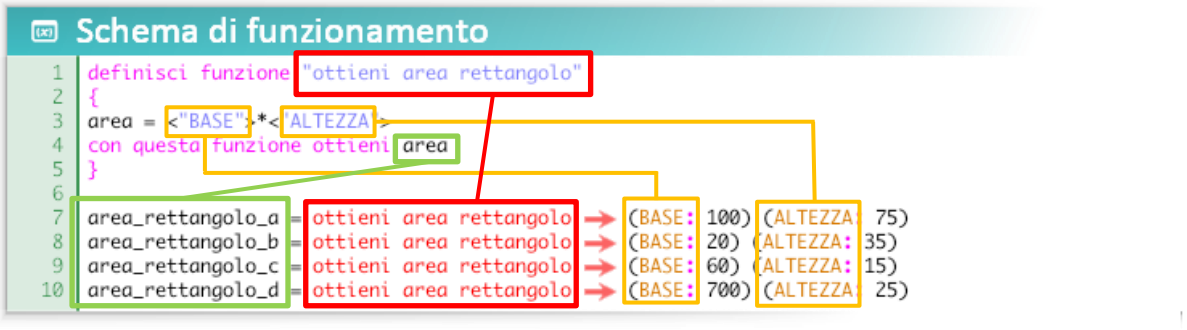
- il nome della funzione deve iniziare con "**ottieni**"
- il codice tra le graffe deve contenere il costrutto "**con questa funzione ottieni <espressione>**"

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:

Esempio

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

Le variabili personalizzate all’interno del costrutto “definisci funzione” possono funzionare in due modi diversi:

- se la variabile è dichiarata all’interno del costrutto verrà considerata **locale alla funzione** e quindi potrà assumere valori diversi per ogni invocazione, senza influenzare le invocazioni precedenti o successive.
- Se la variabile è dichiarata al di fuori del costrutto verrà considerata **globale** e quindi la sua modifica influenzerà ogni parte del codice in cui viene utilizzata, **come avviene normalmente**.

Una variabile predefinita non può mai essere, in alcun modo, considerata locale ad una funzione.

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.

SPECIFICARE ARGOMENTI PER LE PROPRIE FUNZIONI

Esplicitare gli argomenti di una nuova funzione non è fondamentale per il suo funzionamento. Tuttavia è sempre buona norma farlo, principalmente per due motivi:

- se un argomento non viene specificato e in seguito non viene utilizzato nell'invocazione di una funzione, il suo valore predefinito sarà 0. In alcuni casi può rivelarsi un problema.

- Se gli argomenti non vengono specificati non compare il suggerimento in basso nell'editor: per una funzione personalizzata **è fondamentale fare in modo che l'utilizzatore capisca il suo utilizzo**, soprattutto se la definizione della funzione è inclusa dall'esterno.

Per esplicitare gli argomenti bisogna utilizzare il costrutto "**argomenti**" La sintassi da utilizzare è la seguente:

```
argomenti (etichetta: valore predefinito) (etichetta: valore predefinito) ...
```

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

```
definisci funzione "nome funzione"
{
argomenti (etichetta: valore predefinito) (etichetta: valore predefinito) ...
... codice ...
}
```

Esempio

```
1  definisci funzione "disegna due cerchi"
2  {
3  argomenti (X: 400) (Y: 300) (COLORE: rosso)
4  disegna cerchio → (X: <"X">) (Y: <"Y">) (COLORE: <"COLORE">)
5  disegna cerchio → (X: <"X">+500) (Y: <"Y">) (COLORE: <"COLORE">)
6  }
7
8  disegna due cerchi →
```

```
26 definisci funzione "disegna due cerchi"
27 {
28 argomenti (X: 400) (Y: 300) (COLORE: rosso)
29 disegna cerchio → (X: <"X">) (Y: <"Y">) (COLORE: <"COLORE">)
30 disegna cerchio → (X: <"X">+500) (Y: <"Y">) (COLORE: <"COLORE">)
31 }
32
33 disegna due cerchi →
```

Il colore da utilizzare. Può essere una costante (come rosso, blu o giallo) o una variabile che contiene un colore (vedi funzioni ottieni colore rgb/hsv)

disegna due cerchi --> (X:) (Y:) (COLORE:)

Il testo del suggerimento comparirà dopo il simbolo →, come per le funzioni predefinite del linguaggio.

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

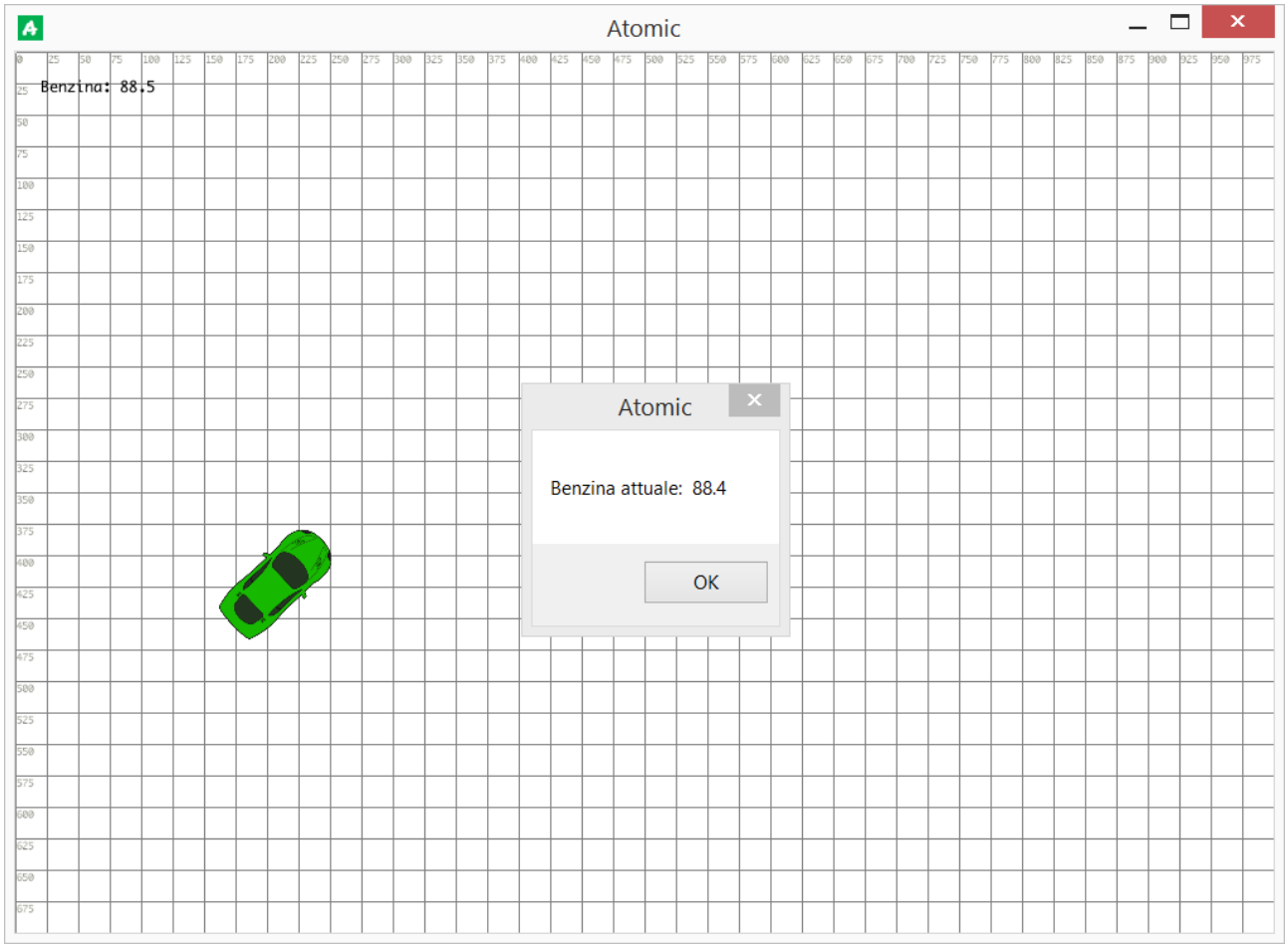
Tramite il costrutto **debug** è possibile fermare momentaneamente l'esecuzione del programma e mostrare un qualsiasi messaggio, anche inerente al codice, ad esempio mostrando il valore di una variabile o il risultato di un'espressione in quel determinato momento. È possibile utilizzare il costrutto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

La sintassi è molto semplice:

```
debug "stringa di testo"
```

Esempio

```
1  //Accellerazione, freno e retromarcia
2  se benzina dell'auto è maggiore di 0
3  {
4      se tasto freccia su è premuto = vero
5      {
6      modifica un elemento → (NOME: auto) (VELOCITA: VELOCITA+accelerazione) (benzina: benzina-0.1)
7      }
8
9      se tasto freccia giu è premuto = vero
10     {
11     modifica un elemento → (NOME: auto) (VELOCITA: VELOCITA-accelerazione*3) (benzina: benzina-0.1)
12     debug "Benzina attuale: <benzina dell'auto>"
13     }
14 }
15
```



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 → (ZE: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: "Il cubo di <numero> è <valore>")

a = 10
b = 3
valore2 = ottieni MELA → (X a) (Y: b)
disegna testo → (TESTO: "la somma 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.

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>
```

Esempio

```
gatto = 4
cane = vero
topo = "Ciao"
```

COMMENTI

Sintassi:

```
//commento su singola linea

/*commento
multilinea*/
```

Esempio

```
//gatto = 4
/*cane = vero
topo = "Ciao"*/
```

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)
```

Esempio

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

Esempio

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

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)
```

Esempio

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

Esempio

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

AUMENTARE/DIMINUIRE UNA VARIABILE

Sintassi:

```
aumenta x di n
diminuisci x di n
```

Esempio

```
1  aumenta orsetto_lavatore di 1
```

Esempio

```
1  diminuisci orsetto_lavatore di 1
```

ISTRUZIONI CONDIZIONALI

se, ripeti finche, ripeti per x volte

Sintassi:

```
C {…}
Condizione { blocco di istruzioni }
```

Sintassi semplificata:

```
C {…}
Condizione allora blocco di istruzioni .
```

Esempio

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

Esempio

```
1  se cane = 2 allora disegna cerchio .
2  finche cane < 2 allora disegna cerchio .
3  ripeti per 3 volte : disegna cerchio .
```

OPERATORI LOGICI E DI CONFRONTO

NOME	VERO NOME	"SIMBOLO INFORMATICO"	SIMBOLO MATEMATICO
non	NOT	!	¬
e	AND	&&	∧
o	OR	||	v
o esclusivamente	XOR	^^	⊕

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

LEGGERE LE VARIABILI LOCALI DEGLI OGGETTI

Sintassi:

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

Esempio

```
bersaglio del giocatore
benzina dell'auto
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.

Esempio

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

ESEGUIRE CODICE ALL'INTERNO DI ELEMENTI

Sintassi:

```
elemento esegue al suo interno questo codice  { … }
```

Esempio

```
INIZIA
crea un oggetto → (NOME: pallina) (valore: 10) (tinta: 0)

CICLO CONTINUO
se tasto sinistro del mouse è stato premuto
{
crea un esemplare → (OGGETTO: pallina) (X: x del mouse) (Y: y del mouse)
}

pallina esegue al suo interno questo codice
{
aumenta X di valore
fattore_casuale = ottieni un valore compreso tra questi → (VALORE 1: 0.1) (VALORE 2: 5)
se X > larghezza finestra { valore = -10*fattore_casuale }
se X < 0 { valore = 10*fattore_casuale }
aumenta tinta di 1
COLORE = ottieni colore hsv → (TINTA: tinta)
disegna testo → (X: X+25) (Y: Y) (TESTO: "Coordinate: <X>,<Y>; Velocità: <valore>")
disegna cerchio → (X: X) (Y: Y) (COLORE: COLORE) (RAGGIO: 10)
}
```

DEFINIRE NUOVE FUNZIONI

Sintassi:

```
definisci funzione "nome funzione" { … }
```

Oppure con restituzione di un valore:

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

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)
```

Esempio

```
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)
area_rettangolo_c = ottieni area rettangolo → (BASE: 60) (ALTEZZA: 15)
area_rettangolo_d = ottieni area rettangolo → (BASE: 700) (ALTEZZA: 25)

disegna testo → (TESTO: area_rettangolo_a) (Y: 20)
disegna testo → (TESTO: area_rettangolo_b) (Y: 40)
disegna testo → (TESTO: area_rettangolo_c) (Y: 60)
disegna testo → (TESTO: area_rettangolo_d) (Y: 80)
```

INCLUDERE CODICE

Sintassi:

```
includi “…”
```

Esempio

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