

Appunti di scuola

Introduzione alla programmazione

Diomede Mazzone

2020, ver. 0.15



Sommario

Sommario	1
Introduzione alla programmazione	2
Istruzioni	2
Sequenza	2
Selezione	3
Iterazione	3
Variabili e strutture dati	4
Variabili	5
Liste e matrici	5
Procedure e funzioni	6
Algoritmo	7
Formalismo grafico	7
Diagrammi di flusso	7
Kojo	11
Definire una funzione	13
Parametri	16
Istruzioni basilari	17
Gestione dei colori	18
Programmazione concorrente	19
Ricorsione e frattali	20
Twine	24
Interfaccia	24
Costruire una storia	26

Introduzione alla programmazione

Un programma, un software, o una qualunque procedura che ci permette di realizzare una determinata attività è in sostanza una sequenza di operazioni ben determinata e che produce sempre lo stesso risultato, con il vincolo però che le azioni vengano sempre realizzate nello stesso ordine.

Spesso una procedura viene definita per descrivere un'operazione complessa, da comunicare ad un altro soggetto oppure per istruire una macchina. Per evitare ambiguità nell'interpretazione della procedura descritta, è importante definire un **linguaggio formale** attraverso il quale codificare le istruzioni che si stanno illustrando. I linguaggi formali, inoltre, possono essere di varia natura, sia grafici che testuali.

Sarà possibile, quindi, definire un programma come una procedura espressa in un linguaggio che descriva l'attività necessaria alla realizzazione di qualunque tipo di “prodotto”, senza ambiguità e con un livello di dettaglio tale da essere sempre riproducibile. Questo prodotto può essere indifferentemente una pietanza da cucinare oppure un software applicativo.

Istruzioni

All'interno di una procedura si parla di istruzione quando si identifica una determinata attività da dover svolgere con precisione e senza la quale non è possibile eseguire l'istruzione successiva. Si parla di non ambiguità perché un calcolatore, che in genere è l'esecutore di un software, non è dotato di intelletto e quindi non può interpretare un'istruzione che gli viene fornita. Ad esempio, se si dovesse istruire un calcolatore per realizzare una pietanza, non si dovrebbero usare istruzioni del tipo “un pizzico di sale” oppure “sale quanto basta”, ma si dovrebbe indicare la quantità di sale in modo preciso ed universalmente riconoscibile.

Le istruzioni quindi possono essere considerate come articolazioni più o meno complesse di tre tipi di attività che si identificano nel seguente modo:

- Sequenza
- Selezione
- Iterazione

Sequenza

La sequenza non è altro che un insieme di istruzioni poste in un determinato ordine che non potrà essere mescolato. Si noti, quindi, che se vengono indicate le istruzioni A B e C in questo

ordine, l'istruzione C non potrà essere eseguita se prima non verrà conclusa l'esecuzione dell'Istruzione B, eseguita comunque dopo la conclusione dell'Istruzione A.

Ogni elemento di una sequenza può essere semplice o complesso, perché si potrebbe racchiudere in un'unica istruzione il frutto di un'articolazione complessa di attività più semplici.

Selezione

La selezione è una struttura di controllo che permette la scelta di un percorso in alternativa ad un altro. Volendo rappresentare un algoritmo attraverso una rappresentazione visiva, potremmo paragonarlo ad un percorso per le biglie. Lasciata cadere una biglia dall'alto, all'interno della pista, può svolgere un unico percorso e se esistono possibili biforcazioni potrà percorrere soltanto una strada, in funzione di eventuali condizioni che di volta in volta possono mutare. È immaginabile quindi che in una struttura di selezione siano presenti più rami, ma l'algoritmo può evolvere solo attraverso uno di questi in funzione di parametri che il programmatore avrà ipotizzato e definito. L'esecutore, infatti, è come una biglia, non è in grado di scegliere una strada da percorrere ma intraprenderà il percorso che in qualche modo gli viene vincolato.

Bisogna quindi caratterizzare una struttura di selezione attraverso una condizione logica che quando si valuta può essere vera o falsa, così da permettere all'esecutore di intraprendere la strada A oppure la strada B. In tal modo è possibile intuire che la parola chiave di una struttura di selezione è **SE** che generalmente viene definita come **IF** mutuando dall'inglese. L'altra parola chiave è **ALLORA (THEN)** attraverso cui è possibile indicare cosa accade se risulta vera la condizione che accompagna il SE. La terza parola chiave di una struttura di selezione è **ALTRIMENTI (ELSE)**, attraverso cui si indica l'alternativa al SE.

In breve il costrutto si può riassumere come IF - THEN - ELSE, secondo l'esempio seguente.

```
IF (la pasta è insipida) THEN metti sale  
IF (c'è il sole) THEN metti gli occhiali ELSE porta l'ombrello
```

Iterazione

Una struttura iterativa, detta anche ciclo, permette di rappresentare operazioni che devono essere ripetute più di una volta, sempre nella stessa sequenza. In ogni tipo di rappresentazione delle procedure, non ha senso ripetere più volte lo stesso blocco che rappresenta un'istruzione. Se bisogna suggerire ad un amico di lanciare una palla tre volte non ha senso dire: "lancia la palla, lancia la palla, lancia la palla", verrà detto invece "lancia la palla 3 volte". Questo tipo di

struttura, quindi, permette di rappresentare ripetizioni di sequenze di istruzioni, ottimizzando spazio e tempo.

Le ripetizioni di istruzioni possono essere di varia natura, principalmente è possibile definire due tipologie:

- Un'istruzione che deve essere eseguita un determinato numero di volte, come nell'esempio descritto in precedenza. Questo tipo di ciclo in genere viene rappresentato nella maggior parte dei linguaggi di programmazione con la parola chiave **FOR**.

Per ripetere sei volte una operazione attraverso un ciclo di FOR, la sintassi potrebbe essere questa:

```
FOR ( i = 0; i<6; i=i+1) { operazioni da ripetere }
```

- Un'istruzione che deve ripetersi **FINCHÈ** non si verifica una condizione. Questo tipo di ciclo viene rappresentato solitamente con la parola chiave **WHILE**.

```
WHILE (condizione da verificare) { operazioni da ripetere }
```

La sintassi relativa a questi due cicli verrà approfondita nel momento in cui si affronterà la programmazione relativa alle diverse rappresentazioni formali di un algoritmo.

Variabili e strutture dati

Le istruzioni di un programma in generale agiscono sui dati che hanno la stessa importanza delle istruzioni, quando si ragiona in modo procedurale. Un dato è un'informazione che viene elaborata oppure valutata a seconda dei casi.

Se si vuole ipotizzare la procedura che gestisce un distributore automatico di bibite, i dati su cui lavora sono relativi alle informazioni dei prodotti che deve distribuire, dei codici per la loro selezione e del relativo costo. A volte le informazioni sono invece dati da valutare, ad esempio una soglia da superare o non superare, come potrebbe essere un voto oltre il quale aver raggiunto la sufficienza o al di sotto del quale non averla raggiunta. È intuibile quindi come il modo di conservare e manipolare queste informazioni risulti estremamente importante per il buon funzionamento di una procedura, per questo motivo bisogna introdurre il concetto di **variabile**. Si tenga presente inoltre che un dato deve rappresentare sempre un'informazione, perché se non ha un significato riconoscibile perde di senso.

Variabili

Una variabile è un contenitore nel quale viene conservata un'informazione. Esattamente come un contenitore fisico, una variabile può contenere un'informazione alla volta sostituendola quando gli si attribuisce un nuovo valore.

Si immagini un bicchiere vuoto, se si decide di conservare dell'acqua lo si riempie con dell'acqua, se si decide successivamente di conservare una bevanda gassata, si sostituirà l'acqua con la bevanda gassata, perdendo il liquido che conteneva.

Riassumendo le caratteristiche di una variabile, si avranno due elementi distintivi:

- **NOME.** Deve avere sempre un nome significativo perchè deve far intendere il significato del valore contenuto. Se il risultato di una procedura è l'area del rettangolo, è buona norma conservare il valore della base all'interno di una variabile che si chiama **Base** ed il valore dell'altezza all'interno di una variabile che si chiama **Altezza**. In genere tutti i nomi sono validi anche se bisognerebbe rispettare alcune convenzioni, come ad esempio non iniziare il nome di una variabile con un numero. Alcune regole invece sono vincolanti, non è possibile infatti inserire spazi all'interno di un nome.
- **VALORE.** Ogni variabile deve contenere un valore. Alcuni linguaggi di programmazione prevedono la dichiarazione di variabili all'inizio del programma, anche se non si conosce il valore da inserire al suo interno. In questi casi il valore dichiarato inizialmente è quasi sempre 0.

Altro aspetto fondamentale delle variabili è la caratterizzazione del tipo di valore che possono contenere. Da questo punto di vista possiamo distinguere due tipi di variabili, **semplici** o **strutturate**. Il tipo di dato che la variabile può contenere dipende dal significato che deve assumere e dall'uso che ne deve essere fatto. Se bisogna calcolare l'area di un poligono, ad esempio, le variabili che conterranno i valori della base e dell'altezza saranno necessariamente di tipo intero o al massimo reale se il valore numerico è di tipo decimale. Se invece bisogna elaborare un testo, la variabile che dovrà contenerlo sarà di tipo stringa. Le tipologie di dati ed il loro utilizzo verranno approfondite successivamente e dipendono dal linguaggio di programmazione che comunque condividono la maggior parte dei tipi possibili.

Liste e matrici

Quando si presenta la necessità di conservare più informazioni all'interno di una stessa variabile allora si è in presenza di variabili di tipo strutturato. Ad esempio se bisogna utilizzare un contenitore per conservare i nomi dei mesi dell'anno, non basterà più una variabile semplice, che potrà contenerne al massimo uno, ma si dovrà considerare una variabile che ad un unico nome

associa 12 elementi. Una variabile strutturata di questo tipo prende il nome di **lista** (o array) che permette di identificare con un nome una struttura fatta da più elementi, tutti identificabili attraverso un indice ordinato. Per l'esempio dei mesi, quindi, alla posizione 3 (mese[3]) corrisponderà Marzo, alla posizione 1 corrisponderà Gennaio e così via. Se una variabile è stata paragonata ad un bicchiere, si potrebbe identificare la lista come un vassoio all'interno del quale sono allineati su una riga un numero di bicchieri pari al numero di elementi appartenenti alla lista. in un linguaggio come Kojo, questo tipo di strutture dati prende il nome di collezione, ma ogni linguaggio di programmazione può associargli anche altri tipi di nomi.

Procedure e funzioni

È stato introdotto il concetto di procedura facendo riferimento ad una sequenza di istruzioni volte alla risoluzione di un compito. In realtà un compito complesso può essere scomposto in più sotto problemi, si Immagini ad esempio lo chef in una cucina. Non sarà lui a provvedere alle preparazioni di tutti gli elementi che gli servono per cucinare, ma delegherà ai suoi aiutanti parte dei preparati che gli serviranno per comporre il suo piatto. Ogni aiutante, quindi, realizzerà una procedura autonomamente e restituirà il prodotto ottenuto allo chef in modo tale che si possa completare il prodotto finale.

Per quanto detto si può dire che la procedura principale potrebbe non utilizzare semplici istruzioni, ma identificare istruzioni più complesse all'interno delle quali sono previste procedure autonome, magari da richiamare più volte in più punti della procedura principale. A questo punto, considerando la possibilità che più procedure possano cooperare, risulta utile distinguere il programma **chiamante** dal programma **chiamato**. Il primo invoca il secondo e attenderà la sua conclusione, appena il programma chiamato terminerà la sua esecuzione restituirà il controllo al programma chiamante. Volendo continuare l'esempio precedente, si potrebbe dedurre che appena il collaboratore dello chef avrà terminato le sue operazioni darà la possibilità allo chef di continuare la sua procedura.

Un approccio alla programmazione di questo tipo si chiama **approccio modulare**, perché permette di dividere le attività in moduli, così da poterle richiamare ogni qualvolta le si ritiene utili. Talvolta però queste attività hanno bisogno di avere valori da elaborare, ad esempio l'aiuto cuoco ha bisogno delle uova per preparare una frittata, oppure una funzione geometrica ha bisogno dei valori della base e dell'altezza per ricavare l'area del poligono. I valori che vengono passati ad una procedura per poter funzionare si chiamano **parametri** o **argomenti** della procedura.

I parametri di input sono quei valori che servono alla procedura per evolvere, senza i quali non potrà ottenere alcun risultato. In alcuni casi, però, i parametri possono essere anche di output,

perché la procedura ha bisogno di restituire un valore al programma chiamante. In questo caso la procedura prende il nome di **funzione**.

In Kojo, per far comprendere al calcolatore che vogliamo definire una procedura o una funzione bisogna utilizzare la parola chiave **def**, ma ogni linguaggio programmazione ha la propria sintassi per definire una nuova procedura. Si tenga presente che una funzione può arbitrariamente avere o non avere sia parametri di input che di output, in funzione delle necessità.

Algoritmo

Sono stati definiti nei paragrafi precedenti gli elementi formali, strutture di controllo, procedure e funzioni, per descrivere in modo efficace un'attività senza ambiguità e con un livello sufficiente di precisione. Questi elementi **sono gli strumenti cognitivi del pensiero computazionale** attraverso il quale è possibile impiegarli in ogni ambito risulti utile applicare un ragionamento logico avente come obiettivo la soluzione di un problema.

Gli esempi citati, infatti, non riguardavano esclusivamente problemi di matematica o di logica, ma problemi riguardante tutti gli ambiti, per tale motivo non si deve ritenere questo tipo di attività come esclusiva pertinenza dell'informatica.

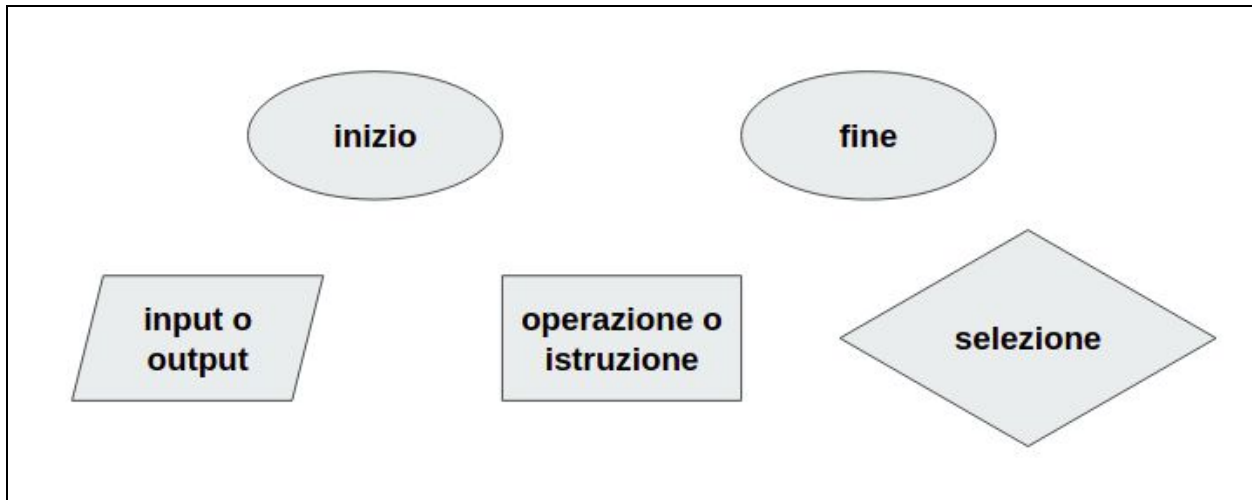
Sarà possibile quindi introdurre il concetto di algoritmo, cioè **una particolare sequenza di azioni, controlli e procedure con la quale si esprime e modella la soluzione di un determinato problema, qualunque esso sia.**

Formalismo grafico

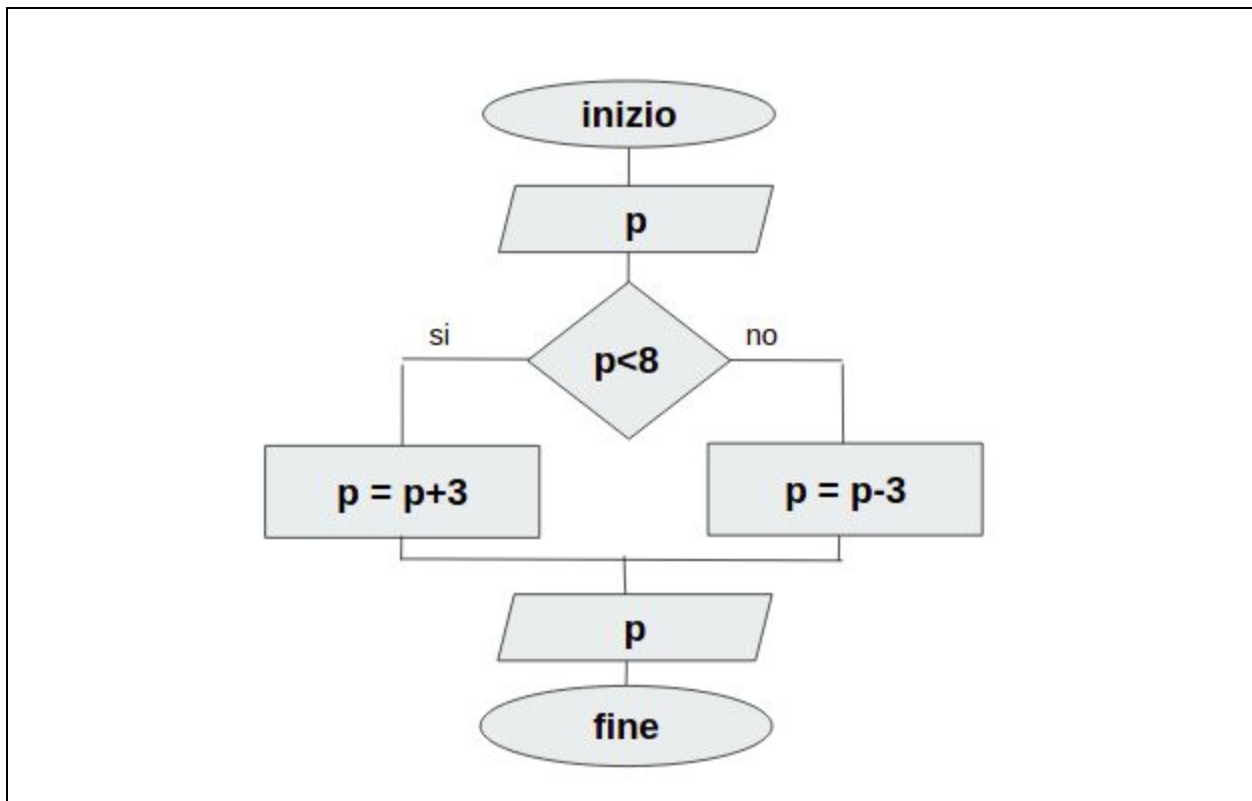
Per rappresentare un algoritmo risulta utile quindi rispettare delle convenzioni sintattiche così da non lasciare ambiguità di interpretazione nella lettura dello stesso. Uno dei formalismi più diffusi e comodi per la condivisione di piccole procedure è il formalismo grafico, in particolare riguardante i diagrammi di flusso. risulta inoltre utile accennare all'esistenza di un altro metodo grafico di rappresentare procedure complesse che viene denominato UML (Unified Modeling Language).

Diagrammi di flusso

Un diagramma di flusso rappresenta, attraverso dei blocchi geometrici, le istruzioni da eseguire in modo sequenziale, dall'alto verso il basso. Inizia sempre con un blocco di tipo "Inizio" e si conclude in un altro blocco ovale di conclusione di tipo "fine".



I blocchi di tipo input o output (parallelogrammo), rappresentano le istruzioni che permettono di acquisire un'informazione dall'esterno attraverso un parametro di input oppure di restituire verso l'esterno un valore attraverso un parametro di output. Il blocco rettangolare, invece, permette di rappresentare un'istruzione semplice, come l'assegnazione di un valore ad una variabile, oppure gruppi di istruzioni richiamando un'altra procedura. Attraverso un blocco di tipo rombo si rende possibile la selezione di un percorso in alternativa ad un altro, all'interno del blocco verrà inserita la condizione da valutare per la scelta del ramo da seguire. Nell'esempio che segue è rappresentata una procedura che applica una struttura di controllo di tipo selezione.



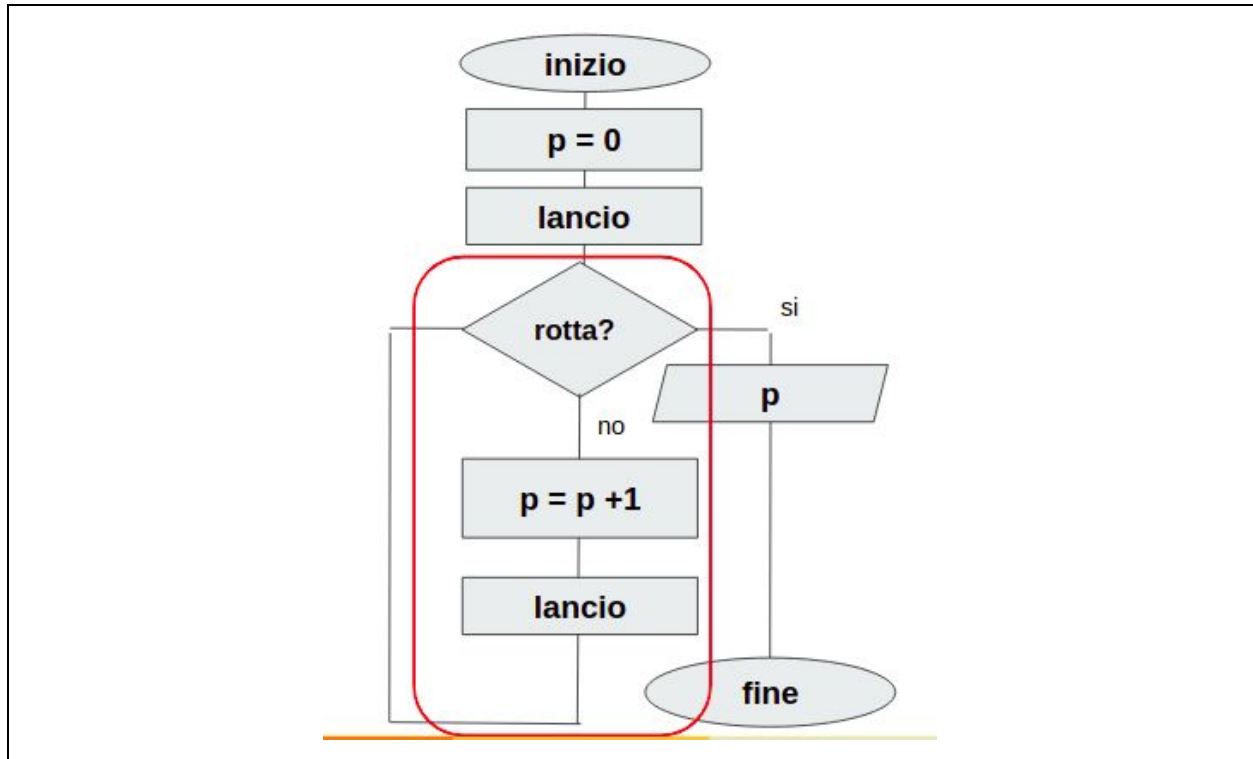
Si noti come la condizione all'interno del blocco di selezione possa essere o vera o falsa, senza alcuna ambiguità. L'esecutore potrà soltanto verificare quella condizione e di conseguenza effettuare le operazioni successive. All'interno dei blocchi rettangolo vengono eseguite istruzioni di **assegnazione**, cioè viene sostituito il valore contenuto nella variabile 'p' con il valore contenuto sommando (o) sottraendo il valore 3, così da aggiornarne il contenuto.

Nell'esempio che segue è possibile vedere la rappresentazione di una struttura di controllo di tipo ciclica. In particolare possiamo riconoscere una condizione di tipo booleano (vengono definite così le variabili in cui è possibile assumere solo valori VERO o FALSO), che se fallisce (risulta quindi FALSA) porta alla fine del ciclo, non ripetendo più i blocchi di istruzione all'interno del ciclo stesso. Questo tipo di struttura rappresenta un ciclo che precedentemente è stato chiamato di WHILE, perché la condizione verificata non definisce un numero preciso di ripetizioni del ciclo, ma dipende da una variabile che in qualunque momento può cambiare stato.

L'esempio riportato rappresenta la soluzione ad un problema definito nel seguente modo:

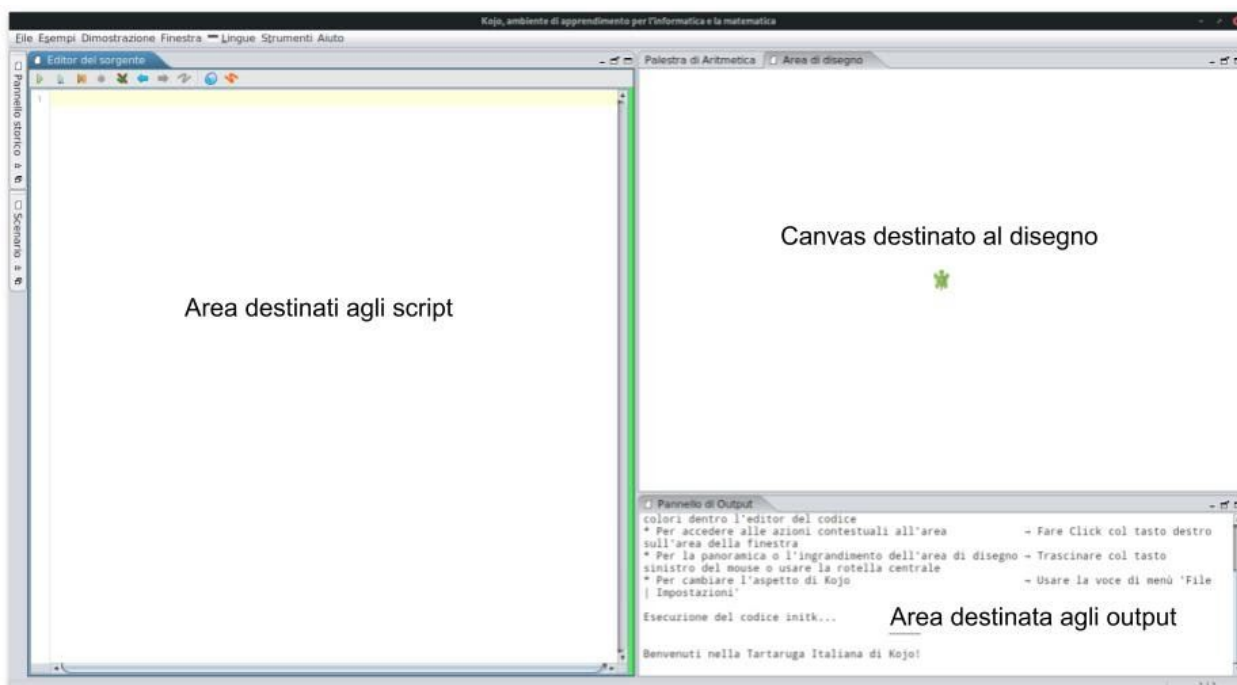
Sia dato un grattacielo dal quale lanciare una lampadina a partire dal piano terra e verificare da quale piano la lampadina si rompe.

Si noti che i primi due blocchi istruzione sono uguali per tipologie ma differenti per struttura, perché il primo assegna un valore ad una variabile che si chiama '*p*', il secondo richiama una ipotetica procedura che si chiama *lancio*.



Kojo

Kojo è un'applicazione che permette di avvicinarsi alla sintassi legata alla programmazione di un calcolatore attraverso elementi molto semplici. Scaricata l'applicazione dal web si noterà che questa prevede più aree, in particolare una sulla sinistra all'interno della quale si inserisce il codice ed una sulla destra all'interno della quale si vede il risultato dell'esecuzione di quel codice.



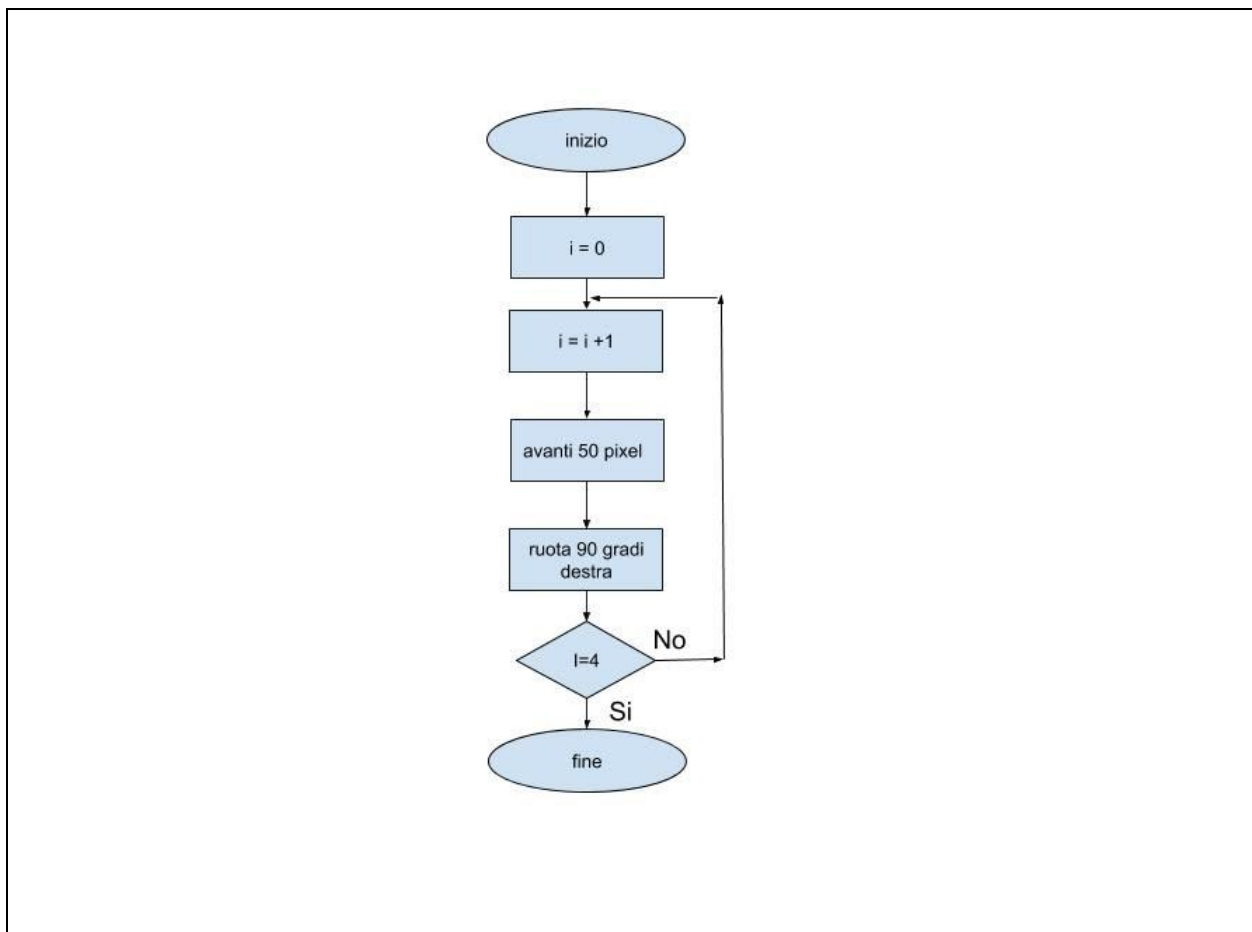
Attraverso il menù principale è possibile accedere a molti esempi, divisi per gradi di difficoltà, verranno trattati in questo testo solo gli esempi cardine per capirne il funzionamento logico.

Si immagini quindi di disegnare un quadrato, volendo descrivere tale procedura in un linguaggio informale si potrebbe scrivere:

- vai avanti
- gira a destra
- vai avanti
- gira a destra
- vai avanti
- gira a destra
- vai avanti

- gira a destra

Proprio perché è stato utilizzato un linguaggio informale, questo si presta a molte ambiguità. Non è stato infatti indicata un'unità di misura per comprendere quanto andare avanti e neppure un angolo di rotazione nel girare a destra. Volendo utilizzare un diagramma di flusso per rappresentare in modo più preciso il disegno di un quadrato, anche attraverso l'utilizzo di cicli, si dovrebbe rappresentare questo algoritmo nel modo che segue.



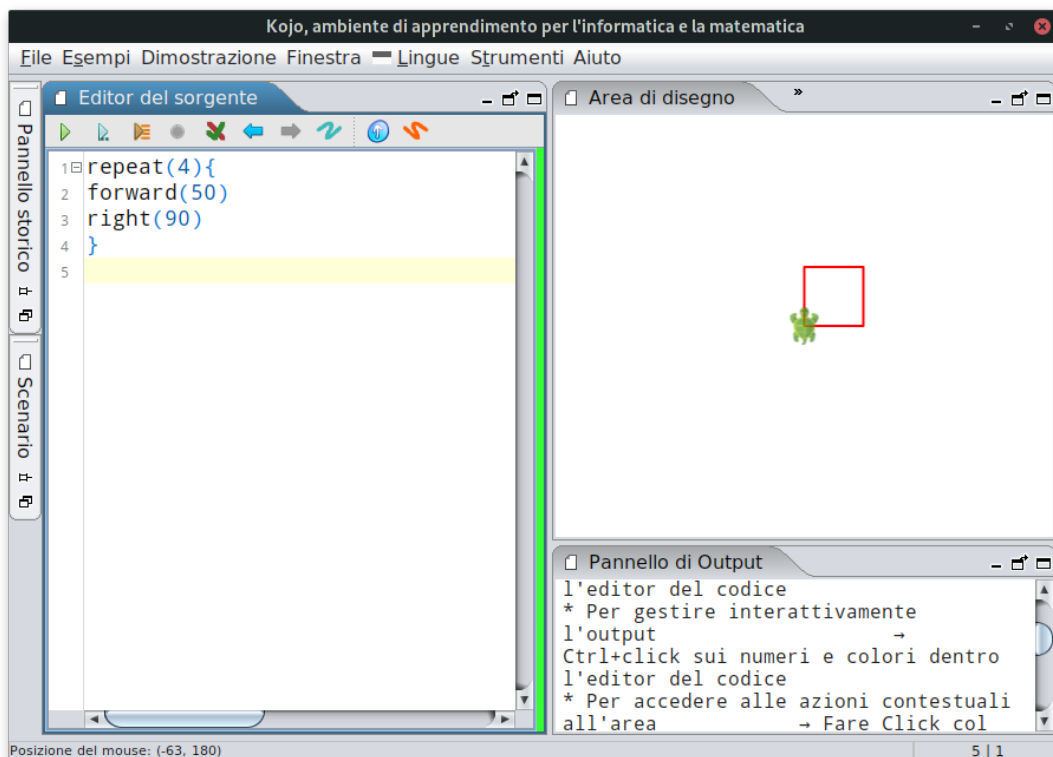
Si noti che in questo caso la struttura ciclica pone la sua condizione di verifica dopo aver svolto almeno una volta le operazioni eventualmente da ripetere. In Kojo, il diagramma di flusso precedentemente descritto, può essere codificato nel seguente modo:

```
repeat(4) {  
  forward(50)  
  right(90)  
}
```

```
}
```

Si noti che le parentesi graffe identificano un blocco, in questo caso il blocco di istruzioni deve essere ripetuto 4 volte, valore indicato come parametro all'interno delle parentesi tonde relative all'istruzione **repeat**.

A questo punto la tartaruga, all'interno dell'area di disegno, realizzerà un quadrato.



Definire una funzione

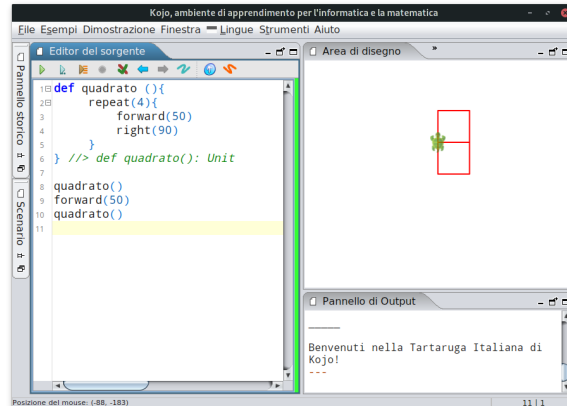
Se volessimo utilizzare le istruzioni precedenti più volte, così da disegnare più quadrati, risulta opportuno inserire quelle istruzioni all'interno di una funzione chiamata *quadrato*. Per definire tale funzione bisogna utilizzare l'identificativo **def**, seguito dal nome della funzione e dalle parentesi graffe all'interno delle quali inserire tutti i blocchi relativi a questa funzione.

È buona norma aggiungere degli spazi all'inizio prima delle istruzioni appartenenti allo stesso blocco, questo migliora la leggibilità del codice, un esempio è rappresentato nell'esercizio che segue. In Kojo, tuttavia, non è errore mettere tutte le istruzioni allineate a sinistra, come accade in altri linguaggi programmazione, ad esempio Python.

Nel codice seguente è definita la funzione `quadrato` ed è richiamata due volte all'interno dello script, infatti se si definisce la funzione e non la si richiama all'interno dell'area di scrittura del codice, questa rimarrà definita ma non chiamata. Si potrebbe quindi definire l'area di lavoro all'interno della quale si inserisce il codice come l'area all'interno della quale si deve inserire il codice del programma **principale** chiamante, prima del quale vanno inserite tutte le funzioni eventualmente da chiamare.

```
def quadrato () {
    repeat (4) {
        forward(50)
        right(90)
    }
}

quadrato()
forward(50)
quadrato()
```



Per identificare un gruppo di istruzioni raggruppabili in procedura oppure in semplici cicli, si consiglia di provare a scrivere il codice in modo strettamente sequenziale ed identificare gruppi identici di istruzioni. Si osservi il codice nell'esempio che segue:

```
clear()
setSpeed(slow)
setPenThickness(4)
setPenColor(red)

var lato = 50

setFillColor(blue)
forward(lato)
right(90)
forward(lato/2)
right(90)
forward(lato)
left(90)

forward(lato)
right(90)
```

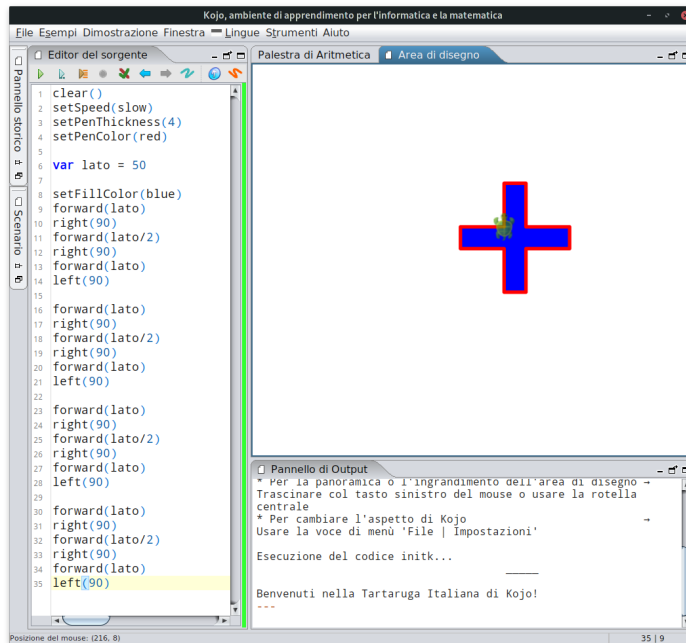
```

forward(lato/2)
right(90)
forward(lato)
left(90)

forward(lato)
right(90)
forward(lato/2)
right(90)
forward(lato)
left(90)

forward(lato)
right(90)
forward(lato/2)
right(90)
forward(lato)
left(90)

```



Nel codice descritto risulta evidente come ci sia un gruppo di istruzioni che si ripete più volte, perchè ogni “ramo” della figura è uguale, ma ruotato. Questo tipo di codice, per semplificare la lettura ed ottimizzare il codice, rispettando un approccio modulare potrà essere scritto in due modi diversi. In un esercizio come quello precedente il consiglio è creare un ciclo con il blocco di istruzioni che si ripetono, successivamente definire una funzione che realizzi quello stesso blocco di istruzioni. Funzione che verrà richiamata nel ciclo della funzione principale.

raggruppando in ciclo	raggruppando in una funzione eseguita in un ciclo
<pre> clear() setSpeed(slow) setPenThickness(4) setPenColor(red) setFillColor(blue) var lato = 50 repeat(4){ forward(lato) right(90) forward(lato/2) right(90) </pre>	<pre> def ramo(){ forward(lato) right(90) forward(lato/2) right(90) forward(lato) left(90) } clear() </pre>

<pre> forward(lato) left(90) } </pre>	<pre> setSpeed(slow) setPenThickness(4) setPenColor(red) setFillColor(blue) var lato = 50 repeat(4){ ramo } </pre>
---------------------------------------	--

Parametri

È stato introdotto dal punto di vista teorico il concetto di parametro, è utile quindi utilizzarlo per definire meglio la funzione quadrato. Può risultare utile infatti personalizzare la lunghezza del lato del quadrato senza dover riscrivere un'altra funzione quadrato con un lato diverso. Per fare questo utilizzeremo un parametro di input da passare alla funzione appena creata all'interno delle parentesi tonde.

```

def quadrato (lato :Int){
    repeat(4){
        forward(lato)
        right(90)
    }
}

clear()
quadrato(30)
forward(50)
quadrato(20)

```

The screenshot shows the Kojol IDE with the following code in the editor:

```

def quadrato (lato :Int){
    repeat(4){
        forward(lato)
        right(90)
    }
}

clear()
quadrato(30)
forward(50)
quadrato(20)

```

The output window displays the execution log, including the initialization of the turtle and the execution of the 'quadrato' function. The drawing area shows a green turtle cursor and a red square drawn on the canvas.

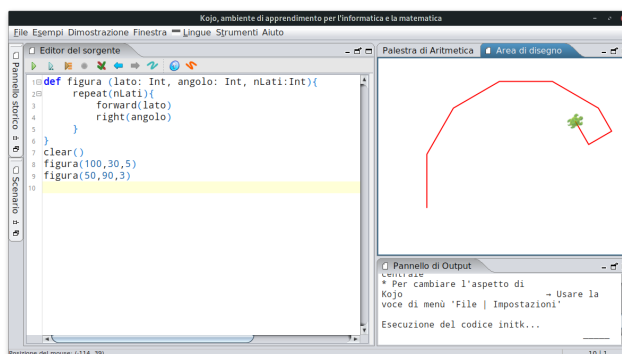
Si noti, nel codice precedente, che all'interno delle parentesi tonde non solo è indicato il nome del parametro (lato), ma è stato anche indicato il tipo di valore da utilizzare, in questo caso un tipo numerico intero (Int).

Nel programma chiamante, la funzione `quadrato` viene richiamata passando il valore numerico direttamente all'interno delle parentesi tonde, sarà *Kojo* ad interpretare quel valore come numero intero da attribuire al parametro di nome *lato*.

È possibile intuire come sia utile talvolta passare alla funzione chiamata più parametri, basterà nella sua definizione elencare i parametri di input, separati da una virgola, all'interno delle parentesi tonde. Quando verrà chiamata la funzione, ovviamente, i valori dovranno essere passati nello stesso ordine con cui sono stati dichiarati.

Nel seguente esempio si definisce una funzione **figura**, che prende in input tre parametri, il primo che rappresenta la lunghezza del lato, il secondo la rotazione dell'angolo ed il terzo il numero di lati da realizzare. Nella funzione chiamante quindi ogni qualvolta viene richiamata la funzione in `figura`, si dovrà inserire tre valori numerici di tipo intero che *Kojo* provvederà a collocare nei rispettivi parametri di input, rispettando l'ordine di scrittura.

```
def figura (lato: Int, angolo: Int, nLati:Int){
  repeat(nLati){
    forward(lato)
    right(angolo)
  }
}
clear()
figura(100,30,5)
figura(50,90,3)
```



Istruzioni basilari

Ogni linguaggio di programmazione prevede una serie di funzioni basilari, integrate nel sistema, che permettono di realizzare operazioni più articolate ma di uso comune. In *Kojo* esistono diversi approcci alla programmazione: **Turtle Graphics**, **Picture Graphics**, **Gaming**, **Robotics**. Tali approcci hanno una progressiva difficoltà nella realizzazione di codice, in questo testo verrà trattato il gruppo di istruzioni associate al primo livello: **Turtle Graphics**.

È possibile riassumere le tipologie di istruzioni secondo la tabella seguente:

Gruppo	descrizione
position	La posizione della tartaruga nell'area di lavoro. Tale punto nel piano

	viene identificato con una coppia, x e y, a partire dall'angolo in basso a sinistra.
heading	La direzione che punta la testa della tartaruga che può assumere un orientamento a 0°, 90°, 180° e 270°, andando in senso antiorario a partire dalla direzione che guarda il lato sinistro dello schermo.
pen up/down	Se la penna è "down" vorrà dire che sta scrivendo, altrimenti non lascia segni sul piano di lavoro.
pen color	Definisce il colore della penna sul foglio di lavoro
pen thickness	Il tipo di linea disegnata nell'area di lavoro
fill color	Il colore dell'area racchiusa dalla linea nell'area di lavoro.

Per quanto riguarda le possibili istruzioni si faccia riferimento alla documentazione on line di Kojo:
<https://docs.kogics.net/reference/turtle.html>

Gestione dei colori

Per quanto riguarda i colori, è bene sottolineare che vengono gestiti con codifica esadecimale, vuol dire che ogni colore è associato ad una combinazione di colori **Red Green Blue**, dove ognuno di questi elementi è un numero compreso tra 0 e 255, in forma esadecimale.

Si ricorda che una codifica esadecimale prevede l'uso di 15 simboli: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, A, B, C, D, E, F, come si potrà dedurre la A corrisponde ad 11 e così via fino a F che rappresenta 15. Ad esempio, se si vuole rappresentare il colore bianco dobbiamo associare 255 a tutti e tre i colori quindi: 255, 255, 255 che in esadecimale diventa: #FFFFFF. Il cancelletto permette sistema di comprendere la codifica.

L'interpretazione della coppia avviene nel seguente modo:

$$FF = (15 * 16) + 15 \rightarrow 255$$

$$C5 = (12 * 16) + 5 \rightarrow 197$$

$$00 = (0 * 16) + 0 \rightarrow 0$$

$$FFC500 = \text{RGB}(255, 197, 0)$$

FFC500

I colori possono essere anche identificati dalla nomenclatura inglese, compreso le sfumature (ad esempio *darkMagenta*), ma in rete sarà possibile trovare convertitori automatici per rintracciare il colore scelto.

per quanto riguarda tutti i possibili utilizzi riguardo i colori si faccia riferimento alla pagina di documentazione dedicata <https://docs.kogics.net/concepts/colors.html>.

Programmazione concorrente

Per introdurre il concetto di programmazione concorrente bisogna soffermarsi sulla definizione di processo e di thread.

Un **processo** è un programma in esecuzione, quindi un qualunque software che viene avviato è un processo che evolve e che viene dato in pasto all'esecutore. L'esecuzione del browser, ad esempio, è un processo che impegnerà una determinata quantità di risorse di sistema. Nei fatti per evolvere un browser ha bisogno di memoria per conservare i dati relativi alla navigazione e di tempo di utilizzo del processore per eseguire tutte le funzioni necessarie alla navigazione. Molte di queste funzioni però potrebbero essere avviate non in una stretta sequenza, ma in concorrenza tra loro. Si pensi alle varie schede del browser all'interno delle quali vengono aperte pagine indipendenti tra loro. È intuibile, quindi, che il processo relativo al browser può essere visto come un contenitore all'interno del quale evolvono in modo autonomo diversi elementi, ciascuno dei quali relativo ad una singola scheda. Questi elementi (che in realtà sono degli algoritmi) vengono definiti **thread**, ognuno con una propria vita che inizia e finisce ogni qualvolta viene aperta o chiusa una scheda. Un programma al cui interno vengono eseguiti più 3D in contemporanea viene detto **multithread**.

Secondo la definizione appena data, ogni qualvolta si costruisce un programma si realizza un processo, che può essere monolitico oppure diviso in più sotto-algoritmi che evolvono in parallelo tra loro. Questo tipo di approccio permette al programmatore di scegliere, a secondo del programma che si deve realizzare, se costruire un algoritmo unico oppure di farlo evolvere con più strade parallele decidendo l'ordine con cui dovranno essere eseguiti ed una eventuale relazione tra loro. Se i thread sono indipendenti, invece, l'onere di scegliere l'ordine con cui eseguirli può essere lasciato al sistema operativo all'interno del quale il programma evolverà.

In Kojo, è possibile generare più sotto algoritmi e realizzare un software multithread, semplicemente attivando più tartarughe. Nell'esempio estrapolato da quelli inclusi nell'applicazione, sono presenti più tartarughe che in parallelo realizzano una figura autonomamente.

```
// In this program, we're trying to make a synchronized drawing with
multiple turtles
clear()

// a new command to make squares
// t - the turtle that draws the square
// n - the size of the square
// delay - the turtle's animation delay; this controls the
synchronization effect
// we use runInBackground below to make the turtles run together
def square(t: Turtle, n: Int, delay: Int) = runInBackground {
    t.setAnimationDelay(delay)
    repeat(4) {
        t.forward(n)
        t.right()
    }
}

val t1 = new Turtle(0, 0)
val t2 = new Turtle(-200, 100)
val t3 = new Turtle(250, 100)
val t4 = new Turtle(250, -50)
val t5 = new Turtle(-200, -50)

square(t1, 100, 100)
square(t2, 50, 200)
square(t3, 50, 200)
square(t4, 50, 200)
square(t5, 50, 200)
```

Ricorsione e frattali

La ricorsione è una tecnica di programmazione che permette ad una funzione di richiamare se stessa durante l'esecuzione. Tale tecnica, pur essendo molto efficace in diverse occasioni, non sempre è di facile comprensione e soprattutto può nascondere errori che potrebbero portare al

blocco del sistema. È necessario quindi usarla con cautela, cercando sempre di comprendere se non esista una strada migliore e più agevole per la realizzazione di un algoritmo.

Un elemento cardine della ricorsione è l'esistenza di una **condizione** che ferma il processo di ricorsione, quindi interrompe il richiamo a se stessa durante l'evoluzione e risalire la catena di richiami. Una funzione ricorsiva, infatti, risulta essere **chiamante** e **chiamata** contemporaneamente.

Si analizzi il seguente esempio, estratto dagli esempi inclusi in Kojo, per comprendere il concetto di ricorsione.

```
def tree(distance: Double) {  
  if (distance > 4) {  
    setPenThickness(distance/7)  
    setPenColor(cm.rgb(distance.toInt,  
math.abs(255-distance*3).toInt, 125))  
    forward(distance)  
    right(25)  
    tree(distance*0.8-2)  
    left(45)  
    tree(distance-10)  
    right(20)  
    forward(-distance)  
  }  
}  
  
clear()  
setSpeed(fast)  
hop(-200)  
tree(90)
```

Nella prima parte del codice viene definita la funzione **tree** che per essere richiamata ha bisogno di avere in input un valore reale (Double) associato alla variabile *distance*.

La prima istruzione di questa funzione è la condizione di arresto della ricorsione

```
if (distance > 4) {
```

In tal modo nella catena di richiami nel momento in cui *distance* diventerà un valore minore uguale a 4 la funzione *tree* fermerà la catena ricorsiva in quanto, osservando il codice, non esiste nessun'altra istruzione al di fuori di questo *if*.

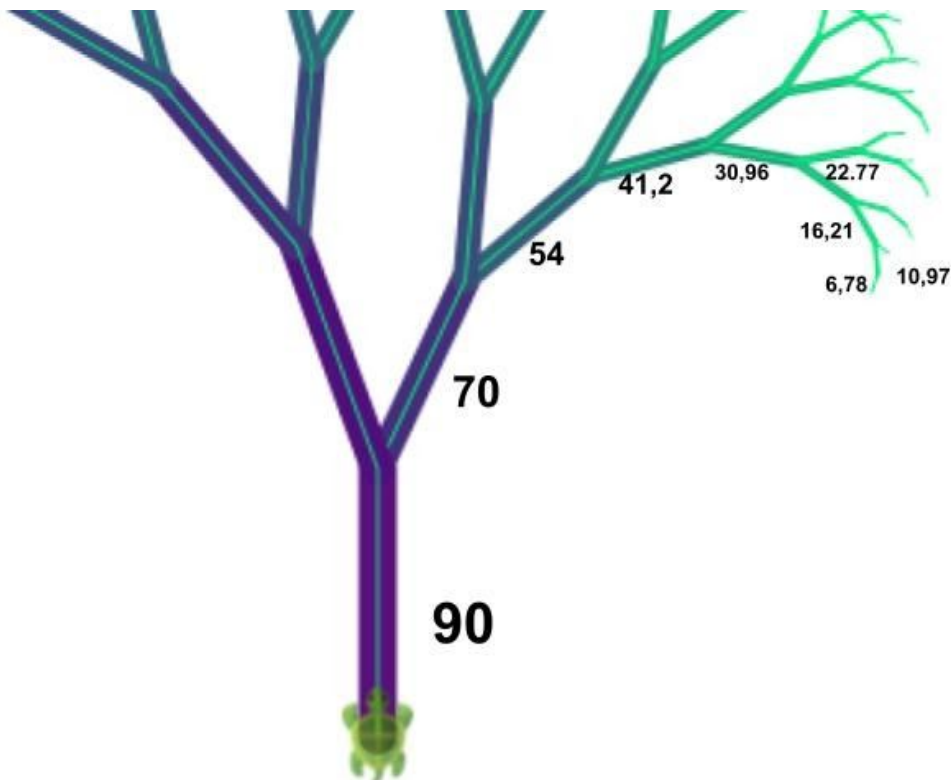
Si noti che il miglior modo per comprendere come avvengono le ricorsioni è osservare il percorso svolto dalla tartaruga a velocità ridotta, impostandola nel programma chiamante definito dopo la funzione *tree*.

Si osservi adesso il codice interno alla funzione.

Imposta lo spessore della penna ad una frazione della distanza	<code>setPenThickness(distance/7)</code>
Imposta il colore del ramo che sta disegnando in funzione della distanza	<code>setPenColor(cm.rgb(distance.toInt, math.abs(255-distance*3).toInt, 125))</code>
Disegni ramo della lunghezza ricevuta in input	<code>forward(distance)</code>
Ruota di 25 ° per iniziare il sotto ramo di destra	<code>right(25)</code>
Richiama se stessa, riducendo i valore della lunghezza acquisito in input	<code>tree(distance*0.8-2)</code>
Finita la ricorsione introdotta con l'istruzione precedente Orienta la penna per disegnare il ramo di sinistra.	<code>left(45)</code>
Richiama la ricorsione decrementando la distanza da disegnare per realizzare il ramo di sinistra.	<code>tree(distance-10)</code>
Ruota verso destra per partire con il sotto ramo di destra dei rami di sinistra.	<code>right(20)</code>
Ripercorre indietro il segmento appena disegnato alla conclusione di tutte le ricorsioni del ramo in	<code>forward(-distance)</code>

oggetto.	
----------	--

Si noti che Il programma chiamante dell'esempio avvia *Tree* passando come parametro della distanza il valore 90. La prima chiamata ricorsiva passerà il valore 70, questa realizzerà la prima biforcazione verso destra. Tale chiamata, a sua volta, richiamerà ricorsivamente la funzione passando un valore pari a 54, realizzando la seconda biforcazione a destra. In questo modo verranno realizzati tutti i rami verso destra fino a che il valore da passare non risulti minore o uguale a 4 secondo l'immagine che segue, in cui ad ogni ramo è affiancato dal valore *distance* con il quale è stato realizzato. Quando il valore di *distance* scende al di sotto di 4, viene eseguita la rotazione verso sinistra e si avvia la ricorsione per la generazione di tutti i sotto rami.



Per quanto riguarda la sintassi è utile notare alcuni aspetti. Ad esempio **distance.toInt** è la conversione del valore contenuto nella variabile *distance* in un numero intero. Il valore di input infatti risultava definito reale, ma per essere passato come valore utile alla definizione di un colore deve diventare di tipo intero. Questa operazione si chiama **Casting**.

Twine

Twine (<https://twinery.org/>) è un ambiente nel quale approcciarsi al coding, anche senza dover necessariamente utilizzare un linguaggio di programmazione nel senso classico del termine. È stato più volte detto che il coding non è semplicemente o unicamente programmazione, per questo motivo è possibile applicarlo nella realizzazione di testi, senza rispettare la linearità classica che verrebbe implicitamente imposta da un classico editor di testo. Quando un testo, di qualunque natura, di fantasia o didattico, sfugge ad una lettura lineare, vuol dire che deve prendere la forma di **Ipertesto**, tipico del web.

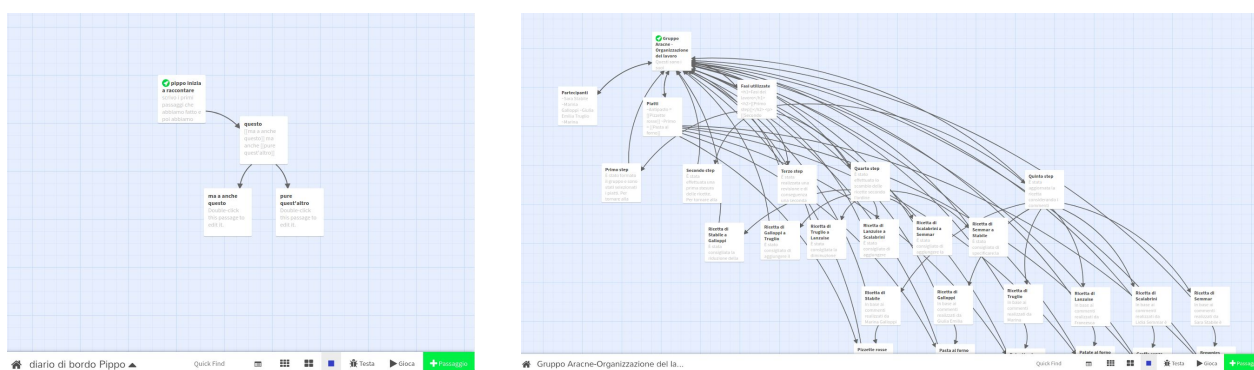
Un testo di questo tipo, che realizza non solo salti, ma anche interazione con l'utente, necessita di strumenti diversi da un semplice editor di testo. Sarà necessario conoscere, quindi, linguaggi detti di **markup**, che non sono linguaggi di programmazione, ma un modo diverso di scrivere testi e parole tale da essere interpretati dal software di lettura in modo corretto. In questo modo sarà possibile formattare l'estetica del lavoro, ma anche realizzare salti e connessioni tra gli elementi del testo. Il più noto linguaggio di markup comprensibile da tutti i Browser è l'HTML. Se, inoltre, un testo html viene integrato con un linguaggio di programmazione come **Javascript**, sarà possibile anche realizzare condizioni di salto da una sezione all'altra rendendo il lavoro particolarmente interattivo, allontanandosi dalla linearità tipica di un lavoro letterario. All'HTML, per la struttura delle pagine ed al Javascript, per una visione più "software" del prodotto finale, va aggiunto l'utilizzo dei fogli di stile CSS, per la personalizzazione dell'estetica che con il puro HTML risulterebbe molto vincolante. Questi aspetti più articolati, comunque, verranno affrontati successivamente.

In questo contesto *Twine*, attraverso una sintassi particolarmente semplice, permette di realizzare testi non lineari, costruendo un prodotto html e javascript, senza necessariamente conoscere la sintassi di questi due linguaggi. Twine può quindi essere considerato uno strumento di **storytelling digitale**.

Interfaccia

Dal sito ufficiale di Twine è possibile installare la versione legata al proprio sistema operativo, ma è anche possibile utilizzare la versione on line, così da non dover installare nulla. Non è possibile lavorarci in contemporanea, come avviene con tutte le applicazioni in Cloud, tuttavia è possibile esportare il file del lavoro svolto, così da condividerli con altri utenti che potranno importarli e modificarli nella propria installazione (o utilizzo on line).

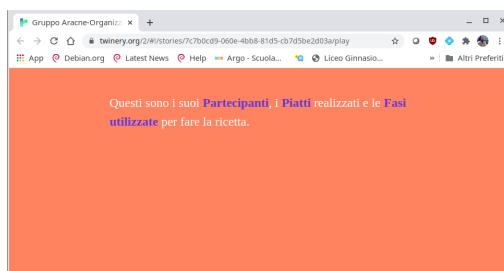
È stato descritto l'output di un lavoro svolto nell'applicazione, ossia contenuti testuali collegati tramite ipertesto, ma non è stato ancora affrontato come viene gestita la produzione di tali contenuti. Bisogna quindi sottolineare che Twine offre una sostanziale differenza di visualizzazione del lavoro in produzione con il lavoro in esecuzione. Per la produzione risulta infatti utile procedere come in una mappa concettuale, quindi con "elementi" posti in connessione tra loro. L'interfaccia di Twine permette di costruire singoli elementi all'interno dei quali scrivere e personalizzare il testo oggetto dell'elemento, visualizzando contemporaneamente le relazioni tra i diversi blocchi. Nelle immagini che seguono è possibile vedere le relazioni tra i blocchi, visualizzati per titoli e collegamenti.



L'interfaccia permette una visualizzazione sintetica degli elementi, fornendo un menù, in basso a destra, utile alla loro gestione e visualizzazione. In particolare il tasto "Gioca" permette di avviare il primo blocco così da iniziare la navigazione esplorando tutti i blocchi, seguendo le scelte dell'utente. Tutti gli aspetti relativi ai tasti previsti ed alle funzionalità integrate verranno rimandate ai paragrafi che seguono.



La visualizzazione dell'output è una pagina web che interpreta il primo blocco, collegato a quelli successivi tramite link di ipertesto, esattamente come avverrebbe in un sito web html.



Costruire una storia

Per iniziare una storia è sufficiente aggiungere il primo elemento, attraverso la pagina principale del software utilizzando il tasto “**+Racconto**” nella barra laterale.



Gli altri tasti principali permettono di:

- **Importa da File.** importare tutti i lavori svolti e archiviati precedentemente.
- **Archivia.** Tutte le storia in un archivio
- **Formati.** Il tipo di storia da scrivere, riguarda la sintassi da usare considerando che ne esistono diverse. In questo testo ne verrà presentata una sola ed è quella di default del software.

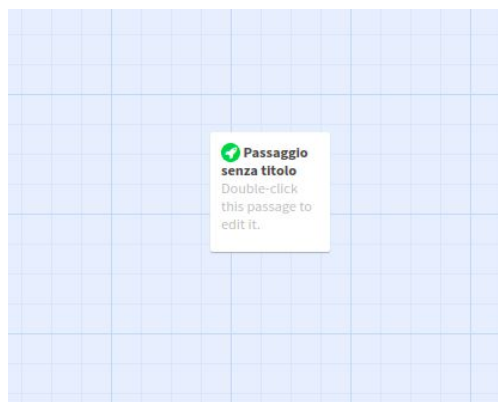
Avviato il racconto viene chiesto il nome, sempre modificabile, così da iniziare a scrivere il primo blocco.



Qual'è il nome del Racconto?
(successivamente potrà essere modificato)

✕ Annulla

+ Aggiungi



Cliccando due volte sul blocco, oppure selezionando la voce modifiche che appare al passaggio del mouse, sarà possibile iniziare a scrivere con un editor di testo semplificato.



Le singole voci del menu rimandano alle strutture classiche di una pagina html, in questa sede non le spiegheremo nel dettaglio perchè risulta molto più utile provare a sperimentare. Risulta invece utile sottolineare le caratteristiche del nodo: Il titolo, le Etichette e la possibilità di creare connessioni ad altri nodi. Basterà, infatti, racchiudere tra doppie parentesi quadre una parole affinché Twine generi un altro nodo e le connessione che arrivano ad esso.



Osservando il risultato ottenuto dal link generato dalle doppie parentesi quadre è evidente come, con l'incremento di altri nodi, si viene a generare una mappa strutturata della storia che si sta costruendo. Se un nodo deve essere raggiunto da più blocchi, sarà sufficiente inserire il nome di quel nodo all'interno di parentesi quadre in tutti i blocchi da cui deve essere raggiunto. È necessario inoltre che il nome risulti essere esattamente corrispondente al blocco da raggiungere, perché la sintassi legata ai nomi è molto vincolata.

Per visualizzare l'anteprima del lavoro svolto basterà cliccare sull'apposito tasto del menu che appare al passaggio del mouse sul singolo blocco. Si avvierà, in questo modo, la lettura delle pagine HTML scritte all'interno del browser che si sta utilizzando.