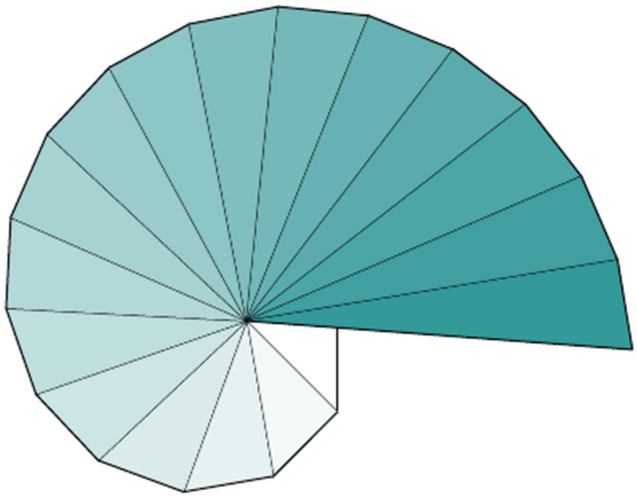
LUIGINO CALVI

# CODING

## UN APPROCCIO AL PENSIERO COMPUTAZIONALE ATTRAVERSO LA ROBOTICA EDUCATIVA



2025

Una *grande* scoperta risolve un grande problema ma c'è un granello di scoperta nella soluzione di ogni problema.
Il tuo problema può essere modesto; ma se esso sfida la tua curiosità e mette in gioco le tue facoltà inventive, e lo risolvi con i tuoi mezzi, puoi sperimentare la tensione e godere del trionfo della scoperta.
Questa esperienza ad una età suscettibile può creare un gusto per il lavoro mentale e lasciare un'impronta nella mente e un carattere per tutta la vita.

G. Polya, *How to solve it*

# Indice

# Presentazione

L'evoluzione tecnologica dei nostri tempi può essere identificata nell'oggetto che chiamiamo genericamente "calcolatore" e più recentemente nell'oggetto "robot". Si tratta di due diverse concretizzazioni del più generale concetto di "esecutore". Tutto ciò ha avuto l'effetto collaterale di distinguere ed evidenziare, di pari passo, i concetti di "solutore", "algoritmo" e "linguaggio". L'algoritmo costituisce il nesso logico e funzionale fra solutore ed esecutore nella catena di comando sintetizzabile nella frase: "il solutore risolve il problema creando un algoritmo che verrà tradotto in un programma che verrà eseguito da un esecutore". La cinghia di trasmissione di questa catena di comando è costituita dal linguaggio mediante il quale si realizza la comunicazione fra solutore ed esecutore.

## I contenuti

Quanto di seguito presentato si sviluppa lungo quattro filoni argomentativi che si intrecciano in un ordito sul quale si regge l'informatica: problemi, algoritmi, macchine e linguaggi.

1. *Problemi*: nei vari capitoli vengono affrontati e risolti diversi problemi di movimento, di interazione, con agganci all'algebra ed alla geometria del piano, in particolare alla "geometria della tartaruga" ed alla "geometria del taxi".
2. *Algoritmi*: per ciascun problema vengono proposti degli algoritmi risolutivi impostati secondo le metodologie top-down e bottom-up. Gli algoritmi sono descritti utilizzando i tipici controlli della programmazione strutturata. Vengono affrontati anche concetti apparentati con la teoria degli algoritmi quali *risorsa* (nel caso specifico coincidente con le azioni elementari svolte), *misura* di una risorsa (coincidente con il numero di azioni) ed *ottimizzazione* (coincidente con la minimizzazione della misura delle risorse impiegate), *equivalenza* (coincidente con l'uguaglianza della misura delle risorse impiegate da due algoritmi).
3. *Macchine*: il termine "macchina" viene solitamente usato per indicare un generico meccanismo che, più o meno automaticamente, svolge delle azioni. A seconda del contesto il termine viene declinato con automa, calcolatore, robot ed altri. Nell'immaginario comune, astraendo da qualsiasi sua realizzazione fisica, un *robot* è un meccanismo autonomo, dotato di un ben specifico insieme di capacità di base, in grado di eseguire delle azioni in un dato ambiente. In ambito didattico lo spazio in cui si muove ed agisce un robot si configura come un ambiente in cui sperimentare strategie di comportamento che, alla fine, vengono concretizzate in algoritmi che hanno un immediato aggancio con il mondo fisico. Una efficace sintesi di tutto ciò è fornita dalla citazione di Goldberg riportata in testa al primo capitolo.
4. *Linguaggi*: tutti gli algoritmi descritti sono orientati e finalizzati alla loro codifica verso il linguaggio `LFR`, un minuscolo e semplice linguaggio di pro-

grammazione in grado di gestire gli automi Quadretto e Puntino, e verso un tradizionale linguaggio di programmazione come Python o C. Si noterà che il linguaggio `LFR` è molto limitato e limitante in quanto principalmente orientato alla sola gestione del movimento. Ma proprio per questa sua caratteristica risulta interessante dal punto di vista didattico, permettendo di stimolare quelle doti di inventiva e creatività che si svilupperebbero usando un linguaggio macchina di basso livello, di tipo assembly.

## La robotica educativa

Molti dei contenuti trattati ricadono all'interno di quella che viene denominata *robotica educativa* (*r.e.*), ossia dell'utilizzo dei robot in campo didattico. La r.e. coinvolge diversi aspetti: i robot e le loro capacità, il movimento, il linguaggio per descrivere il movimento, il programma scritto nel dato linguaggio per ottenere l'effetto desiderato, lo spazio in cui si svolge il movimento. Ognuno di questi argomenti permette di avvicinarsi, sebbene in modo parziale, a campi diversi: alcuni tradizionali della matematica (aritmetica, geometria, algebra) altri considerati inseriti nell'alveo dell'informatica (linguaggi, programmazione, linguistica, algoritmi), altri ancora nella tecnologia (robotica, automatismi).

Almeno nella accezione qui proposta, la r.e. non è un (ulteriore) contenuto e tantomeno una tecnologia, ma è essenzialmente una palestra in cui allenare il pensiero e la mente; ed il robot non è altro che lo strumento, il compagno di viaggio, per raggiungere questi obiettivi. Nelle modalità e negli aspetti contenutistici qui presentati, la r.e. si configura come un pretesto per indagare alcuni aspetti che riguardano il pensiero computazionale.

## Gli automi

L'attore principale che incontreremo nei capitoli che seguono è costituito da un automa con stato, in grado di muoversi nel piano e di interagire con entità esterne.

Si inizia al capitolo *Muovere* con un elementare automa (*Puntino*) che si muove sui vertici del piano quadrettato, con possibilità di rotazioni ad angolo retto ed avanzamenti unitari, lasciando una traccia del suo passaggio disegnando una linea. Puntino è la forma più elementare di robot, non ha alcuna consistenza fisica (è puntiforme) ma è dotato di alcuni attributi geometrici (posizione e verso di avanzamento). Al capitolo *Interagire* viene presentato l'automa *Quadretto* che si muove sulle caselle di una scacchiera ed è in grado di relazionarsi con l'ambiente circostante e con gli oggetti (muri e blocchi) in esso presenti. La limitatezza dell'ambiente in cui si muove Quadretto non deve essere vista come una limitazione ma come un'opportunità di provare e gestire situazioni al limite; offre la tranquillità di agire in un mondo completamente dominabile, con confini facilmente raggiungibili. Al capitolo *Tracciare* si incontra un automa simile a Puntino, ma più evoluto: si tratta della *Tartaruga* dell'omonima geometria che si muove liberamente su tutto il piano geometrico, con rotazioni ed avanzamenti generici. Nonostante l'estrema semplicità, questi robot permettono di affrontare alcune tematiche tipiche della programmazione

e della robotica. Inoltre, offre interessanti situazioni per l'applicazione delle metodologie top-down e bottom-up.

## I capitoli

Nei capitoli che seguono vengono affrontati alcuni aspetti della r.e.; in particolare vengono analizzati gli aspetti logici ed algoritmici, senza entrare nelle questioni tecnologiche; i contenuti descritti costituiscono un'introduzione al pensiero computazionale, al problem-solving, al coding, con incursioni nei linguaggi, nella robotica e nella geometria.

Nel capitolo 1 (*Muovere*) viene presentato un elementare automa, detto *Puntino*, che si muove sul reticolo del piano di coordinate intere, tracciando delle linee secondo la tradizionale impostazione della geometria della tartaruga. Puntino è in grado di avanzare di un numero intero di passi e di ruotare a sinistra o a destra di un angolo retto. Per muovere questo automa viene utilizzato un elementare linguaggio di programmazione testuale (`LFR`). In parallelo alla descrizione di questo linguaggio, vengono introdotte alcune semplici tematiche relative ai linguaggi formali.

Nel capitolo 2 (*Interagire*) si approfondiscono due importanti caratteristiche dei robot (oltre a quella di movimento, già descritta nel cap. 1) : la capacità di *sentire* e di *agire* nell'ambiente. Viene presentato un elementare automa, detto *Quadretto*, che si muove sulle caselle di una scacchiera; similmente a Puntino, questo robot è in grado di avanzare di un passo alla volta, corrispondente ad una casella; inoltre è in grado di ruotare (a sinistra ed a destra) di un angolo retto. Quadretto è dotato di sensori che gli permettono di interagire con altri oggetti presenti sulla scacchiera (muri e blocchi). Per poter gestire efficacemente i sensori, la parte del linguaggio `LFR` già presentata nel cap. 1 viene estesa con dei meccanismi di controllo delle azioni usando delle condizioni. Questo automa può essere preso, per molti aspetti, a paradigma di un generico robot.

La gestione di automi dotati di sensori richiede il ricorso ad algoritmi articolati che non si riesce a codificare mediante il semplice linguaggio `LFR`. A questo scopo, nel capitolo 3 (*Esplorare*), viene utilizzato un pseudolinguaggio testuale dotato della maggior parte delle caratteristiche di un moderno linguaggio di programmazione. Come conseguenza gli algoritmi descritti si prestano ad essere facilmente codificati in un reale linguaggio di programmazione, come Python o C.

Nel capitolo 4 (*Tracciare*) viene presentato l'automa *Tartaruga* che costituisce una variazione evolutiva dell'automa Puntino; Tartaruga è in grado di avanzare di una generica quantità, anche non intera, ed è in grado di ruotare di angoli generici, non necessariamente retti. In altri termini si passa dal piano discreto di Puntino (costituito dai soli punti di coordinate intere) al piano continuo tipico della cosiddetta geometria della tartaruga.

*Nota.* Tutti gli algoritmi presentati nei vari capitoli possono essere provati al computer mediante l'applicazione *AlgMath* liberamente scaricabile dal sito `github/algmath/lab`.

# 1

# MUOVERE

*Gli algoritmi che si incontrano in robotica sono astrazioni che descrivono atti di movimento e di percezione che, quando eseguiti nel mondo reale, consentono di raggiungere obiettivi definiti in termini di oggetti fisici.*

K. Goldberg ed altri,
*Algorithmic Foundation of Robotics*

Il camminare è una delle capacità sviluppate per prime dai bambini, in quanto si svolge in modo spontaneo, senza richiedere la maturazione di avanzate esperienze né l'acquisizione di concetti astratti. Il camminare non ha bisogno di numeri e tantomeno di calcoli: richiede solo di essere in grado di fare un passo alla volta, di cambiare direzione e di avere una strategia di avanzamento. Nei bambini la *volontà* di camminare, il *procedimento* del camminare e l'*azione* del camminare sono assommate ed indistinte. Nell'ambito della programmazione, e più in particolare nel campo della robotica, queste tre componenti vengono distinte ed analizzate singolarmente: la volontà di camminare nasce dall'esigenza di risolvere un problema che richiede dei movimenti, il procedimento di come camminare viene descritto mediante un algoritmo ideato dal solutore e l'azione del camminare viene delegata ad un'entità separata costituita da un robot o altro generico esecutore che si muove in base ai comandi impartitigli dall'esterno oppure in base ad un procedimento interno precedentemente programmato ed inserito nel robot; in entrambi i casi la responsabilità di come muoversi è tutta esterna al robot. In questa contestualizzazione l'azione propria del camminare viene trasferita ad un'entità esterna che viene comandata; mettendosi nell'ottica del solutore che sovrintende al procedimento di movimento, l'azione di camminare viene così più significativamente denominata *muovere*.

In questo capitolo viene presentato un elementare automa che si muove sul piano lasciando una linea di traccia del suo passaggio. Il capitolo costituisce una sorta di introduzione alla cosiddetta *geometria della tartaruga*; data però la particolare semplificazione qui assunta per l'automa (movimento sul reticolo del piano di coordinate intere e rotazioni ad angolo retto), gli aspetti geometrici che si incontrano sono molto blandi e rimangono sullo sfondo; vengono altresì evidenziati alcuni aspetti tipici dell'informatica quali: metodologie di soluzione dei problemi, algoritmi, programmazione, linguaggi e coding.

## 1.1 Il movimento di un automa

Il movimento è una relazione fra due entità; generalmente queste due entità hanno un ruolo asimmetrico: una sta ferma e viene qualificata come *spazio*, l'altra ha un ruolo attivo e *si muove* nello spazio. La relazione fra le due entità spazio e movente è sintetizzata dal concetto di *stato* che, in una situazione semplificata, unisce l'informazione relativa alla posizione ed al verso di avanzamento del movente rispetto allo spazio. Un'entità che si muove in uno spazio in base ad un programma viene spesso denotata con il termine *robot*, indipendentemente dalla sua costituzione fisica.

Nel seguito considereremo una situazione elementare in cui il movimento avviene nel piano. In questo caso, per analizzare il rapporto fra spazio e movente, è utile definire un sistema di riferimento; nel caso di un piano è sufficiente, ad esempio, definire un sistema di assi cartesiani. Un'utile semplificazione del movimento, che ne agevola l'analisi e lo studio, consiste nella sua scomposizione in *sequenza di passi elementari*; è quello che abbiamo fatto da piccoli quando abbiamo imparato a camminare, facendo un passo dopo l'altro. Questa scomposizione passo-dopo-passo del movimento corrisponde ad una discretizzazione del piano in cui ci si muove e del tempo. Nella trattazione semplificata che seguirà il tempo verrà ridotto al concetto di sequenza temporale, in pratica al solo concetto di prima-dopo; estrapolando così il movimento dalla sua contestualizzazione temporale vengono persi alcuni suoi attributi cinematici quali la velocità e l'accelerazione ed il movimento si geometrizza, riducendosi ad una linea.

Consideriamo un automa, ridotto ad un punto, che si muove sui punti di coordinate intere del piano. Denoteremo questo automa con il termine *Puntino*. In ogni istante Puntino ha una posizione sul piano, individuata da una coppia $(x, y)$ di coordinate intere ed un verso di avanzamento $a$ corrispondente ad uno dei quattro punti cardinali; la terna $(x, y, a)$ costituisce lo *stato* di Puntino. Per questo Puntino viene raffigurato mediante un triangolino che ne indica la posizione ed il verso di avanzamento (figura 1.1).

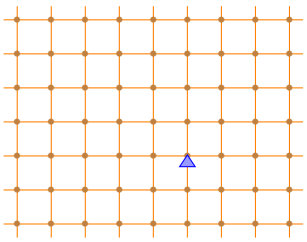


Figura 1.1: Puntino all'interno del piano di movimento; sono evidenziati i punti di coordinate intere del piano in cui può trovarsi.

## 1.2 Le basi del linguaggio LFR

L'esigenza di dover comunicare ad un robot le azioni da intraprendere richiede l'uso di un linguaggio con il quale scrivere i programmi che verranno eseguiti dal robot. Tecnicamente un *linguaggio* è costituito da un insieme di *parole* o *stringhe* formate, rispettando specifiche regole, da sequenze di caratteri di un prefissato insieme finito e non vuoto di caratteri, detto *alfabeto*. Nel caso in cui si tratti di un linguaggio di programmazione, gli elementi del linguaggio vengono detti *espressioni*. In un contesto di programmazione caratterizzato dal paradigma imperativo *solutore-esecutore*, si fa solitamente distinzione fra espressioni che devono essere calcolate (ottenendo un valore come risultato del calcolo) ed espressioni che denotano azioni da eseguire (che producono un effetto come risultato dell'esecuzione); le prime vengono dette *espressioni aritmetiche* e le seconde *espressioni esecutive*. Una sequenza di espressioni esecutive viene detta *programma*.

Nella descrizione di un linguaggio si distinguono solitamente due aspetti:

1. la *sintassi*: descrive le modalità e le regole mediante le quali si costruiscono le frasi del linguaggio o, equivalentemente, si riconoscono le stringhe (corrette) del linguaggio
2. la *semantica*: descrive il significato da attribuire alle stringhe del linguaggio

Nel seguito definiremo un semplice linguaggio finalizzato alla movimentazione dell'automa Puntino; denoteremo tale linguaggio con il termine `LFR` (*L*anguage *F*or *R*obot). Un programma è costituito da una sequenza di espressioni costruite rispettando specifiche regole descritte a seguire. Per semplicità di presentazione verrà adottata una descrizione informale del linguaggio `LFR`, mischiando aspetti relativi alla sintassi con contestuali descrizioni della corrispondente semantica.

### Azioni elementari

Puntino è in grado di eseguire dei movimenti elementari descritti dai valori `F`, `B`, `L`, `R` corrispondenti ciascuno ai seguenti movimenti, relativi alla propria posizione ed al proprio verso di avanzamento:

- `F` : avanzamento di un passo unitario
- `B` : indietreggiamento di un passo unitario
- `L` : rotazione a sinistra di un angolo retto
- `R` : rotazione a destra di un angolo retto

In talune situazioni serve considerare *il movimento nullo*, denotato con $\theta$, che corrisponde a "nessun movimento".

*Esempio 1.2.1* - La forma minimale di espressione è costituita da una singola azione elementare `F`, `B`, `L`, `R`. □

Un'espressione può essere costruita combinando le azioni elementari secondo alcuni schemi prefissati. Considereremo a seguire quattro tipici schemi di programmazione, che consentono di realizzare espressioni articolate ed interessanti: concatenazione, ripetizione, sequenziamento ed inversione.

## Concatenazione di espressioni

I robot più semplici, come Puntino, eseguono le azioni una alla volta, in sequenza. Se $\alpha$ e $\beta$ sono due espressioni, mediante la loro *concatenazione*, scritta nella forma

$$\alpha\beta$$

si ottiene l'espressione costituita dalle azioni di $\alpha$ e di $\beta$ che verranno eseguite nell'ordine indicato: prima quelle di $\alpha$ e poi quelle di $\beta$. La concatenazione di espressioni può essere generalizzata ad una sequenza di espressioni $\alpha_1$, $\alpha_2$, $\ldots$, $\alpha_n$: la scrittura

$$\alpha_1\alpha_2\ldots\alpha_n$$

denota l'espressione costituita dalla concatenazione delle espressioni indicate, che verranno eseguite in sequenza, una dopo l'altra, nell'ordine indicato.

*Esempio 1.2.2* - La concatenazione dell'espressione `FF` con l'espressione `FRFF` produce come risultato l'espressione `FFFRFF` la cui esecuzione ha l'effetto di far avanzare di 3 passi, ruotare a destra ed avanzare di altri 2 passi. □

## Ripetizione di espressioni

Coerentemente con la sua natura di indefesso esecutore, accade spesso che un robot debba eseguire ripetutamente delle azioni elementari o blocchi di azioni. Per queste situazioni i linguaggi di programmazione predispongono dei meccanismi descrittivi che consentono al solutore di indicare la *ripetizione* delle azioni, accorciando la scrittura delle espressioni comprimendole in espressioni più corte.

Se $n$ è un numero naturale ed $\alpha$ è un'espressione, con

$$n\,\alpha$$

si ottiene l'espressione costituita dalla concatenazione di $n$ espressioni $\alpha$, ossia $\alpha\alpha\alpha\ldots\alpha$, $n$ volte $\alpha$. L'operazione di ripetizione ha priorità maggiore rispetto all'operazione di concatenazione; ad esempio, `2FR` corrisponde a `FFR`

*Esempio 1.2.3* - L'espressione `5F` è equivalente all'espressione `FFFFF` che fa avanzare di 5 passi. □

*Esempio 1.2.4* - L'espressione `3FF2R` è equivalente all'espressione `FFFFRR`. □

## Sequenze di espressioni

Se $\alpha_1$, $\alpha_2$, $\ldots$, $\alpha_n$ sono espressioni, con

$$[\alpha_1\,\alpha_2\,\ldots\,\alpha_n]$$

si denota l'espressione formata dalla sequenza delle espressioni $\alpha_i$. L'operazione di sequenziamento risulta necessaria in alcune situazioni, qualora serva delimitare un'espressione come una singola entità, ad esempio per applicare l'operazione di ripetizione oppure per imporre una desiderata priorità di valutazione fra operazioni, ad esempio con l'operazione di inversione che sarà presentata più avanti.

*Esempio 1.2.5* - Raggruppando in sequenza le espressioni `F`, `R`, `FR` si ottiene l'espressione

```
[F R FR]
```

che è equivalente all'espressione `FRFR` che può essere scritta anche `2[FR]`. □

*Esempio 1.2.6* - L'espressione `FFRFFRFFRFFR` muove Puntino lungo un quadrato di 3 unità di lato, ritornando al punto di partenza. Questa espressione può essere compressa nella forma `4[2FR]` che risulta equivalente all'espressione iniziale. □

## Inversione di un'espressione

Per una generica azione $x$ indichiamo con $-x$ l'azione *inversa di* $x$ che, per definizione, è l'azione tale che, eseguita dopo $x$, ripristina lo stato precedente l'esecuzione ripercorrendo a ritroso il percorso fatto mediante $x$. In base a questa definizione, per ogni azione elementare $a$ l'operazione *inversa* è descritta dalla seguente tabella:

| $a$ | $-a$ |
|---|---|
| F | B |
| B | F |
| L | R |
| R | L |

L'operazione di inversione ha priorità maggiore rispetto alla concatenazione ed inferiore rispetto alla ripetizione; ad esempio, `-2FR` corrisponde a `BBR`. Nel caso in cui $\alpha$ sia una sequenza, $-\alpha$ corrisponde alla sequenza inversa avente per elementi gli inversi di $\alpha$; ad esempio, `-[FFR]` = `[-R-F-F]` = `LBB`.

## Costruzione di espressioni

I meccanismi di *concatenazione*, *ripetizione*, *sequenziamento* ed *inversione* possono essere combinati fra loro, in tutti i modi possibili; si possono così ottenere espressioni formate, ad esempio, da "sequenze di ripetizioni di sequenze" e strutture simili. In queste combinazioni, le operazioni di sequenziamento [...] ed inversione − hanno priorità rispetto alle altre operazioni e l'operazione di ripetizione ha priorità rispetto all'operazione di concatenazione.

*Esempio 1.2.7* - Date le priorità degli operatori sopra descritte, l'espressione `2[3F-RF]R` è equivalente all'espressione `2[3FLF]R` che a sua volta è equivalente all'espressione `2[FFFLF]R` che può essere espansa in `FFFLFFFFLFR`. □

## Esecuzione dei espressioni

Un'espressione, per risultare efficace, deve essere eseguita, ossia devono essere eseguite in sequenza temporale le azioni elementari che la compongono, ripetendo l'esecuzione delle parti di espressione che sono moltiplicate in modo prefisso da un valore numerico.

*Esempio 1.2.8 -* L'esecuzione dell'espressione `3[2FR]` fa compiere un percorso a forma di ⊓ con i lati di lunghezza pari a 2 unità. □

## Analisi delle espressioni

*Analizzare* un'espressione significa valutare sulla carta, senza eseguire l'espressione, le principali caratteristiche che emergeranno dalla sua esecuzione. Questa forma di analisi implica capacità di astrazione e previsione, quella tipica che serve ad un progettista di un generico oggetto che deve anticipare con l'idea la sua costruzione ed il suo successivo utilizzo. Le caratteristiche che vengono solitamente considerate nell'analisi di un'espressione sono riportate nelle sotto sezioni che seguono.

***Traccia di un'espressione*** Con *traccia* di un'espressione si intende la sequenza delle azioni elementari generate dall'esecuzione dell'espressione.

*Esempio 1.2.9 -* La traccia dell'espressione `3[2FR]` è costituita dalla sequenza di azioni `FFRFFRFFR`. □

***Equivalenza fra espressioni*** Due espressioni si dicono *equivalenti* (*rispetto alle azioni*) se generano la stessa traccia ossia se la loro esecuzione fa compiere la stessa sequenza di azioni elementari.

*Esempio 1.2.10 -* L'espressione `2[3FR]` è equivalente all'espressione `FFFRFFFR`. □

***Invarianza di un'espressione rispetto allo stato*** Un'espressione è *invariante rispetto allo stato* se la sua esecuzione non modifica lo stato, ossia se lo stato finale raggiunto dopo averla eseguita risulta uguale a quello iniziale.

*Esempio 1.2.11 -* L'espressione `4[2FR]` è invariante rispetto allo stato. □

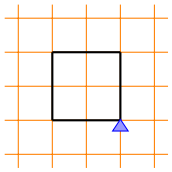
***Complessità di un'espressione*** La *complessità* di un'espressione è data dal numero di operazioni elementari che vengono eseguite nell'esecuzione dell'espressione. Nell'ipotesi che le azioni elementari impieghino lo stesso tempo di esecuzione, la complessità risulta proporzionale al tempo di esecuzione dell'espressione. Date due espressioni equivalenti, per questioni di efficienza, risulta preferibile quella avente la minore complessità.

*Esempio 1.2.12 -* L'espressione `4[2FR]` ha complessità 12, pari alla lunghezza della stringa `FFRFFRFFRFFR` ottenuta espandendo l'espressione in una equivalente stringa piatta. □

## 1.3 Tracciare linee sul piano

Se attacchiamo una penna ad un robot che si muove, come effetto del movimento, viene tracciata una linea che descrive il percorso del robot; otteniamo così uno strumento per disegnare. Spesso, il disegno costituisce proprio l'obiettivo: un problema di movimento diventa un problema di geometria, e viceversa. La corrispondenza fra percorsi e linee costituisce un valido artificio per sviluppare i programmi per movimentare un automa che si muove sul piano. È stata questa l'idea dalla quale si è sviluppata la *grafica della tartaruga*, un approccio alla geometria alternativo alla tradizionale geometria euclidea ed alla geometria analitica, basato sulla movimentazione di un automa. È quanto verrà descritto in questo paragrafo.

*Esempio 1.3.1* - L'esecuzione dell'espressione `4[2FL]` ha l'effetto di disegnare un quadrato avente i lati di 2 unità. La figura che segue, nella quale è riportata anche la posizione ed il verso di avanzamento iniziale (e finale) di Puntino, descrive il percorso generato dall'esecuzione dell'espressione.



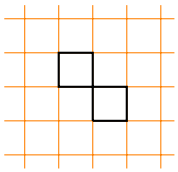
□

La soluzione dei problemi in cui è richiesto il disegno di una figura mediante Puntino può essere impostata secondo i seguenti passi:

1. individuazione della posizione iniziale di Puntino più conveniente
2. individuazione del percorso da realizzare
3. traduzione del percorso mediante un'espressione `LFR`

Questi principi guida sono descritti nella soluzione dei due seguenti problemi.

*Problema 1.3.1* Disegnare la seguente figura:



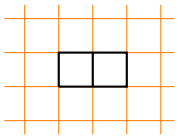
*Soluzione*. La figura può essere vista composta da due quadrati di lato unitario. Il punto più conveniente per la posizione iniziale di Puntino è costituito dal punto di contatto di questi due quadrati. Con questa scelta è

sufficiente disegnare 2 quadrati (`4[FR]`) intervallati da un'azione di raccordo pari ad un'inversione (`2L`); l'espressione completa che ne deriva è la seguente:

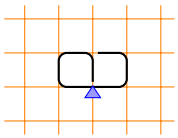
```
2[4[FL]2L]
```

□

*Problema 1.3.2* Disegnare la seguente figura:



*Soluzione*. Nel disegno di figure, per un evidente principio di minimalità, si cerca di muovere Puntino in modo da non ripassare su tratti già disegnati. A questo scopo risulterebbe poco utile vedere la figura composta da 2 quadrati adiacenti e tentare di rifarsi al precedente esempio 1.3.1. Risulta invece più produttivo il percorso evidenziato nella seguente figura dove è riportata anche la posizione di partenza di Puntino:



Questa figura evidenzia due sotto figure: un □ seguito da una ⊐, raccordate fra loro da una rotazione a destra; traducendo in linguaggio `LFR` si l'espressione:

```
4[FL]R3[FL]
```

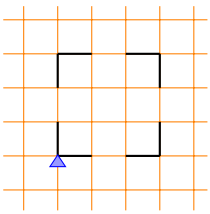
□

Nel problema 1.3.2, con una opportuna scelta del percorso, si è riusciti a tracciare la figura senza ripassare su tratti già disegnati. Non tutte le figure godono però di questa proprietà. Per questi casi è predisposto il seguente comando avente l'effetto di "saltare in avanti", senza lasciare linea di traccia:

`J` : salta in avanti di un passo unitario senza lasciare traccia

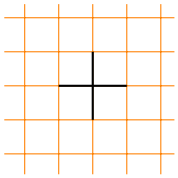
Il comando `J` permette, inoltre, di tracciare figure non connesse. Il comando inverso `-J` viene interpretato come un salto di un passo indietro rispetto al verso di avanzamento corrente ed il comando `-3J` viene interpretato come `-[3J]` che risulta equivalente a `-[JJJ]` e quindi a `-J-J-J`; la sua esecuzione fa fare 3 salti indietro ossia un salto di 3 passi indietro.

*Esempio 1.3.2* - L'esecuzione dell'espressione `4[FJFR]` ha l'effetto di disegnare gli angoli di un quadrato avente i lati di 3 unità, come si vede nella figura che segue.



□

*Problema 1.3.3* Disegnare la seguente figura:



*Soluzione*. In questo problema risulta importante, ai fini di avere una semplice soluzione, individuare il punto di partenza di Puntino più conveniente. In generale, ed anche in questo caso, quando la figura da disegnare presenta una simmetria centrale, conviene posizionare inizialmente Puntino al centro della figura. Per il disegno dei 4 rami della figura, volendo non ripassare su un tratto già disegnato, si vede che, raggiunta l'estremità di un ramo, risulta necessario saltare in centro e da lì, dopo una rotazione, disegnare il successivo ramo. Sulla base di queste considerazioni, la figura richiesta può essere disegnata mediante la seguente espressione:

```
4[F-JL]
```

□

## 1.4 Schemi di scomposizione delle figure

Se il problema consiste nel disegno di una figura allora la metodologia di scomposizione in sottoproblemi si identifica nella scomposizione della figura. Conseguentemente, un'efficace tecnica di problem solving applicata ai problemi di disegno di figure consiste in una preventiva scomposizione delle figure in più parti per poi concentrarsi sul disegno di ciascuna parte. Adottando la metodologia top-down si cerca di scomporre la figura in sotto figure più semplici con la convinzione o speranza di poter facilmente disegnare ciascuna sotto figura; nel caso della metodologia bottom-up si cerca di individuare una scomposizione in sotto figure di cui si conosce già il modo per disegnarle; spesso queste due metodologie vengono usate in modo congiunto.

La scomposizione di figure si fonda essenzialmente su due [1] tipici schemi descritti a seguire, dove con la notazione $\pi(f)$ si denoterà l'espressione che disegna la figura $f$ [2].

- scomposizione *sequenziale*: se $f$ è una figura composta da una sequenza di più parti $f_1$, $f_2$, ..., $f_n$, allora

  $$\pi(f) = \pi(f_1)\, r_1\, \pi(f_2)\, r_2 \ldots r_{n-1}\, \pi(f_n)$$

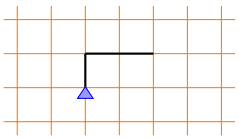
  dove $r_1$, $r_2$, ..., $r_n$ sono delle opportune azioni di raccordo
- scomposizione *iterativa*: se $f$ è una figura composta da una ripetizione di $n$ figure $f_0$ disposte secondo uno schema regolare [3], allora

  $$\pi(f) = n[\pi(f_0)\, r]$$

  dove $r$ è un'opportuna azione di raccordo

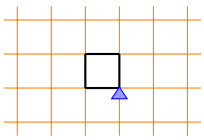
Questi due schemi sono descritti nei due semplici esempi che seguono.

*Esempio 1.4.1* - Consideriamo la seguente figura ad "elle" composta da due tratti di rispettive lunghezze 1 e 2 unità, disposti ad angolo retto:



In questo caso $\pi(f_1)$ = `F`, $\pi(f_2)$ = `2F`, $r$ = `R`, da cui si deduce che l'espressione per disegnare $f$ è $\pi(f)$ = `FR2F`. □

*Esempio 1.4.2* - Disegniamo il seguente quadrato di lato unitario:



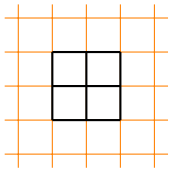
In questo caso $\pi(f_0)$ = `F`, $r$ = `L`, da cui si deduce che l'espressione per disegnare $f$ è $\pi(f)$ = `4[FL]`. □

Naturalmente, si possono avere delle figure più articolate, corrispondenti a combinazioni dei due schemi di sequenzializzazione e ripetizione, come si vedrà a seguire.

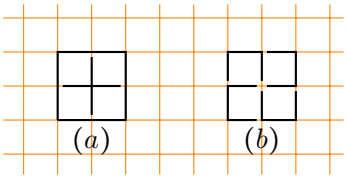
[1] Un altro schema (*ricorsione*) verrà presentato più avanti (al cap. *Interagire*), dove sarà descritto il meccanismo dei *sottoprogrammi*.
[2] Non si tratta di una funzione ma di una corrispondenza in quanto una figura può essere generata da espressioni diverse, anche imponendo delle condizioni di minimalità all'espressione.
[3] La *regolarità* dello schema comporta che l'azione di raccordo $r$ sia sempre la stessa, per ciascuna delle $n$ ripetizioni.

In taluni casi la scomposizione delle figure viene più efficacemente descritta mediante dei formati algoritmici, come descritto nella soluzione del seguente problema.

*Problema 1.4.1* Disegnare la seguente figura:



*Soluzione*. Ad una prima analisi si nota che non si può disegnare la figura mediante un percorso euleriano, ossia senza ripassare su tratti già disegnati e senza salti. La figura può essere scomposta in diversi modi; fra i tanti possibili, risultano interessanti i seguenti due:




Nel caso $(a)$ si evidenzia una scomposizione sequenziale mentre nel caso $(b)$ una scomposizione iterativa. Seguendo la scomposizione $(a)$ si arriva all'algoritmo 1:

**Algoritmo 1** - Disegno di una finestra

1: disegna + centrale ▷ $P_1$
2: salta su un vertice ▷ $P_2$
3: disegna □ esterno ▷ $P_3$

Ci troviamo qui nella situazione di beneficiare della metodologia bottom-up in quanto la soluzione del sottoproblemi $P_1$ e $P_3$ è nota (dal problema 1.3.3 e dall'esempio 1.3.1). Per questioni di simmetria conviene assumere che Puntino inizialmente sia posizionato al centro della figura. Si perviene pertanto facilmente alla traduzione in linguaggio `LFR` come segue:

```
1    4[F-JL]     #croce
2    2[JR]       #salto su un vertice
3    4[2FR]      #quadrato esterno
```

Seguendo invece la scomposizione $(b)$ si arriva, applicando la metodologia top-down, all'algoritmo 2.

**Algoritmo 2** - Disegno di una finestra

**Input:** lunghezza $l$ del lato della croce

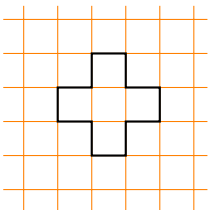
1: **for** 4 **times**
2: disegna $\sqcap$ ▷ $Q_1$
3: salta al centro ▷ $Q_2$
4: ruota $\alpha$ ▷ $Q_3$
5: **end for**

L'immediata soluzione dei tre sottoproblemi $Q_1$ (`3[FR]`), $Q_2$ (`J`) e $Q_3$ (`2L`) porta alla seguente codifica in linguaggio `LFR`:

```
4[3[FR]J2L]
```

□

*Problema 1.4.2* Disegnare la figura a croce riportata a seguire.



*Soluzione*. La struttura ripetitiva della figura suggerisce una struttura ripetitiva della scomposizione in cui $f_0$ è $\sqcap$ e l'azione $r$ di raccordo è una rotazione di un (opportuno) angolo $\alpha$ come descritto nel seguente algoritmo 3.

**Algoritmo 3** - Disegno di una croce

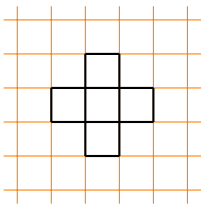
1: **for** 4 **times**
2: disegna $\sqcap$ ▷ $P_1$
3: ruota $\alpha$ ▷ $P_2$
4: **end for**

Le istruzioni 2. e 3. nell'algoritmo 3 costituiscono due sottoproblemi di facile soluzione e di immediata codifica in linguaggio `LFR`: $P_1$ è risolto da `3[FR]` e $P_2$ da `2R`. Ricomponendo le soluzioni dei sottoproblemi $P_1$ e $P_2$ all'interno dell'algoritmo 3 si ottiene la definitiva espressione risolutiva:

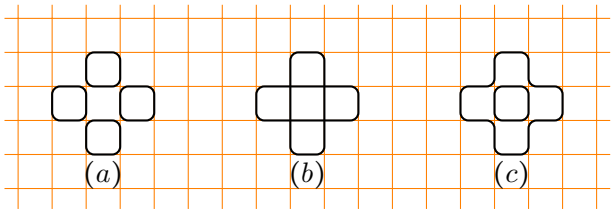
```
4[3[FR]2R]
```

□

*Problema 1.4.3* Disegnare la seguente figura:



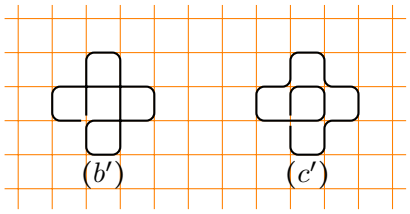
*Soluzione*. Questa figura può essere scomposta in diversi modi, come illustrato nella figura che segue:




Queste diverse scomposizioni riflettono diverse gradazioni di applicazione delle metodologie top-down e bottom-up, come descritto ai punti che seguono.

- $(a)$ la figura viene scomposta in sotto figure uguali (quadrati); questa soluzione assomma una parte di metodologia bottom-up in quanto il disegno di un quadrato è una figura elementare già risolta
- $(b)$ la scomposizione in due rettangoli è una soluzione altamente top-down in quanto il rettangolo è una figura non ancora risolta e richiede quindi un ulteriore passo di analisi per arrivare alla soluzione
- $(c)$ scomposizione altamente bottom-up in quanto sia il quadrato interno e la croce esterna sono delle sotto figure già risolte

A differenza della figura oggetto del precedente problema 1.4.3, la presente figura presenta un'ulteriore complessità in quanto, presentando degli incroci di linee, può essere tracciata secondo diverse traiettorie. In casi come questo, per evidenti questioni di minimalità si cerca di individuare un percorso di tracciamento che non ripassi su tratti già tracciati; un percorso avente questa caratteristica viene detto *euleriano*. In base a questa considerazione, la scomposizione $(a)$ risulta poco utile in quanto non individua un percorso euleriano di tracciamento, mentre la $(b)$ e la $(c)$ individuano rispettivamente i due descritti nella figura che segue:




Scegliendo la scomposizione $(c)$ e la corrispondente traiettoria $(c')$ si ricava che la figura richiesta può essere disegnata mediante la seguente espressione:

```
4[FR]L4[3[FR]2L]
```

□

## 1.5 Assegnazioni

Le assegnazioni costituiscono il meccanismo tipico dei linguaggi di programmazione di tipo imperativo per associare dei nomi mnemonici a delle espressioni. Nel linguaggio `LFR` un'*assegnazione* viene espressa mediante la notazione

$$id \texttt{ = } exp$$

dove *id* è un generico identificatore costituito da una sequenza di caratteri minuscoli ed *exp* denota una generica espressione del linguaggio. L'identificatore *id* viene detto anche *variabile* in quanto uno stesso identificatore può essere assegnato più volte, cambiando quindi il suo valore nel corso dell'esecuzione mentre l'espressione *exp* posta alla destra del segno di assegnazione viene detta detta *r-espressione*. Un insieme di variabili assegnate viene detto *ambiente*. Gli identificatori di variabile possono essere usati come componenti all'interno di un'espressione. Quando in un'espressione si incontra una variabile (non in una *r*-espressione), viene eseguita l'espressione corrispondente al momento alla data variabile.

*Esempio 1.5.1* - Nella linea che segue la variabile $x$ viene assegnata con l'espressione `4[FR]` ed alla linea successiva tale espressione viene eseguita (ottenendo come effetto il disegno di un quadrato di lato unitario).

```
1  x = 4[FR]   #assegnazione alla variabile x
2  x           #esecuzione della variabile x
```

□

*Esempio 1.5.2* - L'esecuzione della seguente porzione di programma comporta l'esecuzione dell'espressione `3[2FR]F`.

```
1  x = 2FR     #assegnazione alla variabile x
2  3xF         #esegue l'espressione 3[2FR]F
```

Si notino le parentesi di raggruppamento nell'espressione `3[2FR]F` che viene eseguita. □

*Esempio 1.5.3* - A seguire è descritto un semplice ambiente costituito dalle variabili $a$, $b$ e $c$:

$$\{(a, \texttt{2F}), (b, \texttt{3[FR]}), (c, \texttt{L})\}$$

□

*Osservazione*. L'esecuzione di un'assegnazione non comporta alcun movimento di Puntino ma solo modifiche all'ambiente (che potrà poi influenzare il movimento). Un'assegnazione viene solitamente denominata *istruzione* e scritta su una linea distinta. Quando un'assegnazione compare all'interno di una sequenza [...] deve essere separata dalle altre espressioni o assegnazioni da una virgola; ad esempio come in `[R,x=Fx,x]`.

*Valutare* un'espressione in un ambiente significa sostituire gli identificatori che vi compaiono con il loro valore attuale, iterando ricorsivamente la sostituzione nel caso le espressioni corrispondenti a qualche identificatore contengano a loro volta altri identificatori.

*Esempio 1.5.4* - A seguire è descritto il processo di valutazione dell'espressione $\texttt{F}z$ nell'ambiente $\{(x, \texttt{2F}), (y, x\texttt{R}), (z, x\texttt{L}y)\}$

$$\texttt{F}z \to \texttt{F}x\texttt{L}y \to \texttt{F2FL}x\texttt{R} \to \texttt{F2FL2FR}$$

□

Un'espressione prima di essere eseguita o assegnata ad un identificatore viene valutata nell'ambiente corrente, usando i valori attuali delle variabili che vi compaiono.

*Esempio 1.5.5* - Nella seguente porzione di programma sono elencate alcune assegnazioni con indicato, come commento a lato, l'evoluzione dell'ambiente a seguito di ciascuna assegnazione. L'esecuzione del programma comporta l'esecuzione dell'espressione `RF2F`.

```
1    a = F        # {(a,F)}
2    x = y2a      # {(a,F),(x,y2F)}
3    y = RF       # {(a,F),(x,y2F),(y,RF)}
4    x            # esegue RF2F
```

□

La scelta degli identificatori, specialmente nei programmi più articolati e complessi, è un fattore importante che influisce sulla leggibilità dei programmi.

*Esempio 1.5.6* - La seguente porzione di programma disegna un quadrato di lato 3 unità.

```
1    ruota = R
2    lato  = 3F
3    quad  = 4[lato ruota]
4    quad
```

La variabile `quad` alla terza linea viene assegnata con l'espressione `4[3FR]` che poi viene eseguita alla linea seguente, generando un quadrato avente il lato di 3 unità. Stesso effetto si otterrebbe sostituendo le ultime due linee con l'espressione `4[lato ruota]`. □

Nel linguaggio `LFR` si ammette che in un'assegnazione la $r$-espressione possa contenere identificatori non ancora assegnati; in questi casi tali identificatori, nel processo di valutazione, vengono mantenuti (*valutazione parziale*); nel caso in cui tutti gli identificatori della $r$-espressione siano stati assegnati, il processo di valutazione è finito e termina producendo come risultato un'espressione che non contiene variabili (*valutazione completa*) [4] . L'uso di una variabile all'interno di una $r$ espressione è una tipica situazione che si verifica quando si fa ricorso alle metodologie top-down e bottom-up.

*Esempio 1.5.7 -* Al termine della seguente sequenza di assegnazioni:

```
1    a = F
2    x = LaRb
3    y = aFx
```

si ottiene l'ambiente $\{(a, \texttt{F}), (x, \texttt{LFRb}), (y, \texttt{FFLFFRb})\}$ □

In una $r$-espressione può comparire l'identificatore della variabile che si sta assegnando. Una tipica situazione si verifica all'interno dei cicli dove risulta possibile cambiare iterativamente il valore di una variabile.

*Esempio 1.5.8 -* Il seguente programma disegna la porzione di spirale quadrata `FR2FR3FR`:

```
1    x = FR
2    3[x,x=Fx]
```

□

Nel caso in cui in un'assegnazione la $r$-espressione contenga l'identificatore che si sta assegnando e tale identificatore non sia stato precedentemente assegnato, la valutazione della $r$-espressione rimane parziale ed il successivo uso della variabile innesca un processo ricorsivo infinito.

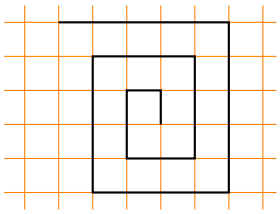
*Esempio 1.5.9 -* Il seguente programma disegna indefinitamente un quadrato di lato 2 unità.

```
1    x = 2FRx
2    x
```

□

[4] Molti linguaggi di programmazione (Python, Java, C, ...) richiedono che le $r$-espressioni contengano tutti identificatori assegnati.

*Problema 1.5.1* Disegnare la porzione di spirale descritta sotto:



*Soluzione*. Dalla figura si nota che i lati della spirale hanno lunghezze $1, 1, 2, 2, 3, 3, 4, 4, 5, 5$. È quindi sufficiente adattare il programma dell'esempio 1.5.8 come segue:

```
1     x = FR
2     4[2x,x=Fx]
3     x
```

□

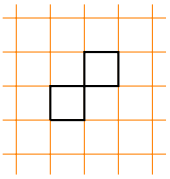
## Assegnazioni e scomposizioni

La metodologia programmativa che sottende al meccanismo delle assegnazioni costituisce un viatico verso il ricorso alla metodologia dei sottoprogrammi, come si vedrà al seguente capitolo. Il ricorso a queste metodologie comporta diversi vantaggi, fra i quali:

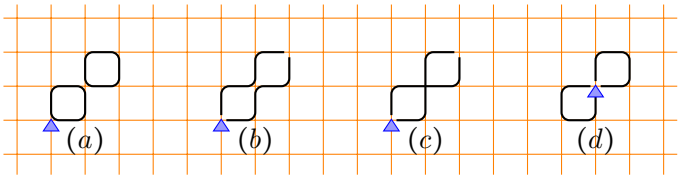
- aumenta la documentazione del codice in quanto espressioni (porzioni di programma) vengono denotate mediante un identificatore che, se ben scelto, descrive sinteticamente quanto svolto dall'espressione stessa
- fornisce un supporto alla metodologia top-down in quanto un sottoproblema si presta ad essere risolto da una corrispondente espressione
- è di supporto alla metodologia bottom-up in quanto favorisce il riuso in più punti di porzioni di codice identificate mediante un nome
- facilita la localizzazione e la correzione degli errori in quanto consente di testare singole porzioni di codice e nel caso di errori è sufficiente correggere solo le porzioni di codice dove la variabile è stata definita.

Tutti questi vantaggi possono essere riconosciuti analizzando le soluzioni dei problemi che seguono.

*Problema 1.5.2* Disegnare la seguente figura:



*Soluzione*. La figura può essere vista e scomposta in diversi modi, come descrive la seguente figura, nella quale è indicato con un triangolino lo stato iniziale di Puntino e sono disegnati gli spigoli arrotondati al fine di meglio evidenziare il percorso:




Le diverse scomposizioni portano, rispettivamente, alle seguenti porzioni di codice, ciascuna delle quali ha l'effetto di disegnare la figura oggetto del problema:

```
#scomposizione (a)
quad  = 4[FR]
salta = JRJL
otto  = quad salta quad
otto
```

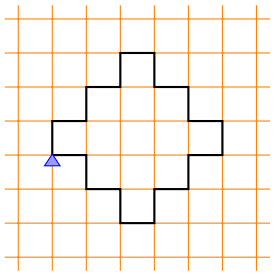
```
#scomposizione (b)
angolo = 2[FR]
scala  = 2[angolo 2L]
otto   = 2[scala 2R]
otto
```

```
#scomposizione (c)
s     = FLF  #angolo a sx
d     = FRF  #angolo a dx
esse = d s
zeta = s d
otto = esse L zeta
otto
```

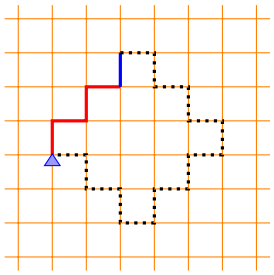
```
#scomposizione (d)
quadrato = 4[FR]
inverti  = 2R
otto     = 2[quadrato inverti]
otto
```

□

*Problema 1.5.3* Disegnare la seguente figura:



*Soluzione*. Partiamo dalla posizione indicata nella precedente figura ed immaginiamo di percorrere delle rampe di gradini ciascuno formato da un tratto verticale (alzata) e da un tratto orizzontale (pedata), con delle opportune rotazioni ad angolo alla fine di ciascun tratto. Complessivamente la cornice risulta composta da 4 rampe, ciascuna composta da 2 gradini con un'alzata di raccordo al termine di ciascuna rampa. La figura che segue illustra una rampa di due gradini (colore rosso) con un'alzata alla fine (colore blu) ed il resto della cornice in tratto punteggiato.

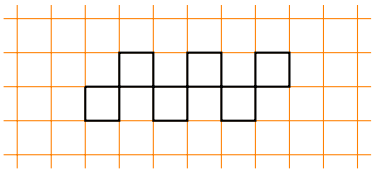


Basandosi sull'analogia dei gradini e sulla scomposizione del percorso come indicato sopra, si perviene facilmente al seguente programma:

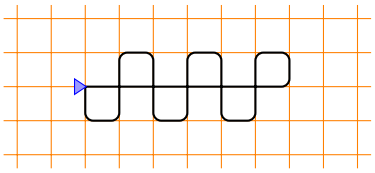
```
1    alzata  = FR
2    pedata  = FL
3    gradino = alzata pedata
4    rampa   = 2 gradino
5    cornice = 4[rampa alzata]
```

Con `cornice` si ottiene l'effetto di disegnare la cornice completa. □

*Problema 1.5.4* Disegnare la seguente figura:



*Soluzione*. Ad un primo approccio la figura appare composta da una successione di 6 quadrati. Questa scomposizione risulta però poco utile in quanto comporta il ridisegno di alcuni tratti al fine di riportarsi sullo spigolo corretto per il disegno del successivo quadrato. Risulta invece più utile la scomposizione evidenziata nella seguente figura dove è indicato con un triangolino lo stato iniziale di Puntino:



Con questa scomposizione si riesce a disegnare la figura senza ripassare su tratti già disegnati e non risulta necessario alcun comando di salto.

In base alle precedenti considerazioni si ricava che la figura può essere disegnata mediante il seguente algoritmo:

**Algoritmo 4** - disegno 6 quadrati

| | |
|---|---|
| 1: tratto lineare verso destra | ▷ `6F` |
| 2: rotazione verso l'alto | ▷ `L` |
| 3: serpentina di ritorno verso sinistra | ▷ $P$ |

Il sottoproblema $P$ viene risolto dal seguente algoritmo:

**Algoritmo 5** - disegno serpentina

| | |
|---|---|
| 1: **for** 3 **times** | |
| 2: percorso ⊓ in senso antiorario | ▷ `2[FL]F` |
| 3: percorso ⊔ in senso orario | ▷ `2[FR]F` |
| 4: **end for** | |

Ricomponendo le soluzioni dei vari sottoproblemi, ossia innestando l'algoritmo 5 alla linea 3 dell'algoritmo 4 si ottiene l'algoritmo completo al quale corrisponde l'espressione

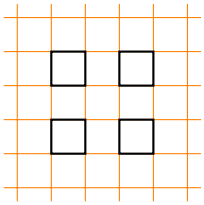
```
6FL3[2[FL]F2[FR]F]
```

Per questioni di leggibilità e per riflettere la struttura dell'algoritmo risulta comunque preferibile spezzare la precedente espressione mediante il ricorso a delle assegnazioni, come riportato a seguire.

```
a = 2[FL]F      #percorso a U antiorario
b = 2[FR]F      #percorso a U orario
s = 3[a b]      #serpentina
t = 6F          #tratto lungo lineare
p = tLs         #percorso completo
p
```

□

*Problema 1.5.5* Disegnare la seguente figura:



*Soluzione*. Una spontanea soluzione si basa sul seguente algoritmo:

**Algoritmo 6** - disegno serpentina

1: **for** 4 **times**
2: disegna un □ ▷ $P_1$
3: salta al successivo □ ▷ $P_2$
4: **end for**

I due problemi $P_1$ e $P_2$ sono fra loro legati in quanto $P_2$ dipende dallo stato finale assunto da Puntino al termine del disegno di un □, e questo dipende dallo stato iniziale. Per sbloccare la situazione supponiamo che Puntino si trovi nel vertice in basso a sinistra del quadrato in basso a sinistra e risolviamo $P_1$ mediante l'assegnazione

```
quad = 4[FR]
```

A questo punto la soluzione di $P_2$ diventa:

```
salta = 3JR
```

L'espressione che disegna la figura si esprime con

```
4[quad salta]
```

□

## 1.6 L'algebra LFR

Ogni contesto algebrico (dei numeri, dei polinomi, ...) viene definito e caratterizzato attraverso le seguenti azioni:

1. specificazione degli elementi dell'insieme su cui si opera (numeri, valori logici, polinomi, ...)
2. definizione delle operazioni elementari sugli elementi dell'insieme (addizione, moltiplicazione, ...)
3. dimostrazione delle proprietà delle operazioni ed eventuale inquadramento in una struttura algebrica generale (monoide, gruppo, ...)
4. definizione dei meccanismi sintattici per combinare le operazioni per ottenere delle espressioni
5. esecuzione di trasformazioni, calcoli e valutazione di espressioni

Tutte queste fasi si possono riscontrare anche nel contesto del linguaggio `LFR` (che chiameremo *algebra LFR*) e sono descritte nelle seguenti sottosezioni.

### Elementi dell'algebra

Gli elementi fondamentali dell'algebra LFR sono `F`, `B`, `L`, `R`, `J`, già descritti nei precedenti paragrafi.

### Operazioni dell'algebra

Le operazioni elementari dell'agebra LFR sono la concatenazione, la ripetizione, la sequenzializzazione e l'inversione, come descritte nel paragrafo 1.2.

### Proprietà delle operazioni

A seguire esamineremo le proprietà algebriche delle operazioni dell'algebra LFR. Queste proprietà verranno utilizzate nelle trasformazioni delle espressioni. Se $\alpha$ e $\beta$ sono due espressioni dell'algebra LFR, per indicare la trasformazione di $\alpha$ in $\beta$ useremo la notazione $\alpha \to \beta$.

***Operazione di concatenazione*** L'operazione di concatenazione gode della proprietà associativa: per ogni espressione $\alpha, \beta, \gamma$ dell'algebra LFR si ha $(\alpha\beta)\gamma = \alpha(\beta\gamma)$ ma non della proprietà commutativa (in generale, $\alpha\beta \neq \beta\alpha$).

*Esempio 1.6.1 -* Usando la proprietà associativa dell'operazione di concatenazione si possono eseguire le seguenti trasformazioni: `LFR2[FR]` → `L(FR2[FR])` → `L3[FR]` □

***Operazione di ripetizione*** L'operazione di ripetizione gode di alcune proprietà che possono essere riassunte come segue: se $m$, $n$ sono due numeri naturali, $x$ un generico movimento elementare ed $\alpha$ una generica espressione, allora

1. è lecito sostituire $n$ caratteri $x$ consecutivi con $nx$
2. è lecito sostituire $n$ espressioni $\alpha$ consecutive con $n[\alpha]$
3. è lecito sostituire $n\alpha\, m\alpha$ con $k\,\alpha$, dove $k = m + n$

*Esempio 1.6.2* - A seguire sono riportate, per ciascuna delle regole sopra elencate, alcune trasformazioni di stringhe:

```
FFRFFF      →  2FR3F
FLFLFRFLFR  →  FL2[FLFR]
2[FLF]3[FLF]  →  5[FLF]
```

□

***Operazione di inversione*** Analizziamo ora il significato e le proprietà dell'operazione unaria `-` di inversione che viene scritta in forma prefissa. Poiché nel linguaggio `LFR` il segno `-` non ha il significato di operatore aritmetico di *opposto* di un numero ma indica l'inverso dell'espressione che segue il segno stesso, la sintassi del linguaggio indirizza questa interpretazione vincolando che a seguire questo segno non ci sia un numero (moltiplicatore di espressione) ma un'azione elementare (`F`, `B`, `L`, `R`, `J`) oppure una sequenza `[...]`; risultano pertanto sintatticamente errate scritture della forma `-2F` che vanno riscritte come `-[2F]`. Per delle generiche espressioni $x$ ed $y$, useremo anche delle scritture della forma $x-y$ che va intesa come un'abbreviazione di $x[-y]$ e non come un'operazione binaria – fra $x$ ed $y$ (che non è definita). Vista poi la priorità dell'operatore prefisso `-` rispetto al numero moltiplicatore prefisso presente eventualmente nell'espressione $y$, la scrittura $x[-y]$ corrisponde a $x[-[y]]$. Ad esempio un'espressione come `5F-[3F]` indica la concatenazione dell'espressione `5F` con l'espressione `-[3F]`. Coerentemente con il contesto algebrico, in un'espressione come `--[2FR]` i due segni `--` (o un numero pari di segni `-`) di doppia inversione possono essere eliminati. Tutte queste osservazioni sono sintetizzate nel seguente elenco di proprietà di facile dimostrazione.

**PROPRIETÀ 1.** Data una generica espressione elementare $\alpha$,

$$--\alpha = \alpha$$

Date le espressioni elementari $\alpha_1, \alpha_2, \ldots, \alpha_n$,

$$-[\alpha_1\, \alpha_2\, \ldots\, \alpha_n] = [-\alpha_n\, -\alpha_{n-1}\, \ldots\, -\alpha_2\, -\alpha_1]$$

Data una generica espressione elementare $\alpha$ ed un numero naturale $k$,

$$-[k\, \alpha] = k\, [-\alpha]$$

□

*Esempio 1.6.3* - Applicando le precedenti proprietà si ha:

```
-[2[FR]] → 2[-[FR]] → 2[-R-F] → 2[LB] → LBLB

-[[FR]2RF] → -F-[2R]-[FR] → B2[-R]-R-F → B2LLB → B3LB
```

□

*Osservazione*. Diversamente da quanto avviene in algebra con la differenza fra due monomi simili, non si possono operare le semplificazioni `5F-[3F]` → `2F` e `3F-[5F]` → `-[2F]` in quanto i movimenti della forma `k`F producono l'effetto collaterale di disegno di una linea.

## Espressioni dell'algebra

Un'espressione può essere considerata come un oggetto matematico, alla stregua di un numero, di una figura o altra entità matematica. Per sottolineare questo aspetto definiremo dapprima la forma delle espressioni del linguaggio `LFR` e successivamente analizzeremo le operazioni e corrispondenti proprietà. Faremo riferimento al sottoinsieme del linguaggio `LFR` descritto nel paragrafo 1.2: si tratta di un linguaggio molto limitato ed inadeguato per gestire molti aspetti della programmazione, ma sufficientemente potente per gestire il movimento di un semplice robot ed interessante per affrontare alcune tematiche tipiche dei linguaggi formali. Questo sottoinsieme del linguaggio ha una struttura statica in quanto non coinvolge condizioni e quindi rimane definita una corrispondenza biunivoca fra un'espressione e la sequenza dei movimenti che vengono generati dalla sua esecuzione.

Un tipico metodo per descrivere un linguaggio, ossia l'insieme delle sue frasi, consiste nell'adottare un *approccio generativo* basato sulla definizione di un insieme di regole mediante le quali generare le frasi del linguaggio. Queste regole costituiscono la *sintassi* (del linguaggio). La sintassi viene definita precisando le parole base costituenti il linguaggio (*assiomi*) e le regole grammaticali (*regole di generazione* o *regole di produzione*) mediante le quali generare, a partire dagli assiomi, le frasi del linguaggio. Nel caso di un linguaggio di programmazione le frasi vengono dette *programmi*. A seguire useremo il termine "programma" come sinonimo di "espressione". Adottando questa tecnica, il linguaggio `LFR` può essere descritto informalmente come segue:

1. *Assiomi*:
   a) ogni carattere dei movimenti `F`, `B`, `J`, `L`, `R`, `O`, `I` è un'espressione
2. *Regole di generazione*:
   b) la concatenazione di due espressioni è un'espressione
   c) è lecito mettere le parentesi `[]` attorno ad un'espressione
   d) è lecito mettere un numero naturale davanti ad un'espressione
   e) è lecito mettere il segno `-` davanti ad un'espressione

La modalità indicata sopra per descrivere un linguaggio è intuitiva ma è inadeguata perché è ambigua e non si presta per alcuni tipi di elaborazione automatica dei programmi. In alternativa si può utilizzare una modalità più criptica e formale, derivata dal linguaggio della matematica, definendo il linguaggio `LFR`, qui indicato con $\mathcal{L}$, come segue:

a') $c \in \{\texttt{F}, \texttt{B}, \texttt{J}, \texttt{L}, \texttt{R}, \texttt{O}, \texttt{I}\} \Rightarrow c \in \mathcal{L}$

b') $\alpha, \beta \in \mathcal{L} \Rightarrow \alpha\beta \in \mathcal{L}$

c') $\alpha \in \mathcal{L} \Rightarrow [\alpha] \in \mathcal{L}$

d') $n \in \mathbb{N}, \alpha \in \mathcal{L} \Rightarrow n\alpha \in \mathcal{L}$

e') $\alpha \in \mathcal{L} \Rightarrow -\alpha \in \mathcal{L}$

Le regole a')-e') consentono di fare un passo in avanti rispetto alle a)-e), ma non è decisivo in quanto vengono mantenute delle ambiguità ed aggiunte delle incoerenze (ad esempio, in questo nuovo formalismo, viene mischiato il concetto di *numero* con quello di *rappresentazione* di un numero). Tutte queste ambiguità vengono risolte ricorrendo alla teoria dei "linguaggi formali".

Una modalità alternativa per descrivere un linguaggio consiste nell'adottare un'impostazione *riconoscitiva* top-down, finalizzata al riconoscimento di una data espressione; per il linguaggio `LFR` si può precisare una lista di specificazioni come segue:

- un *programma* è una lista di istruzioni
- una *lista di istruzioni* è un'istruzione eventualmente seguita da altre espressioni separate da virgola
- un'*istruzione* è un'espressione oppure un'assegnazione
- un*espressione* è una sequenza di termini
- un *termine* è ...

Per descrivere il linguaggio si può `LFR` si potrebbe continuare con la descrizione informale appena iniziata sopra; ma la descrizione di un linguaggio di programmazione ha bisogno di metodi che permettano di esprimere in modo rigoroso e non ambiguo i programmi corretti. Un formalismo spesso utilizzato, detto EBNF (*Extended Backus-Naur Form*), si avvale di simboli variabili (scritti in font *italico*) che costituiscono delle definizioni provvisorie che si risolvono in altre definizioni, fino ad arrivare ai simboli terminali (scritti in font `bold`). La sintassi con la quale viene descritto a seguire il linguaggio `LFR` si conforma parzialmente alla notazione EBNF: il metasimbolo $\longrightarrow$ denota una regola di espansione, ossia una regola in base alla quale la variabile che precede il simbolo può essere sostituita con l'espressione che segue il simbolo, ed il simbolo | denota un'alternativa della regola.

*Prog* $\longrightarrow$ *List*
*List* $\longrightarrow$ *Instr* [ `,` *List* ]
*Instr* $\longrightarrow$ *Exp* | *Assign*
*Exp* $\longrightarrow$ *Term* [ *Exp* ]
*Term* $\longrightarrow$ [ *Minus* ] *Block* | *Num Term*
*Block* $\longrightarrow$ *Move* | *Seq* | *Ident*
*Seq* $\longrightarrow$ `[` *List* `]`
*Minus* $\longrightarrow$ `-` [ *Minus* ]
*Assign* $\longrightarrow$ *Ident* `=` *Exp*
*Num* $\longrightarrow$ *Digit* [ *Num* ]
*Ident* $\longrightarrow$ *Alfa* [ *Alfa* ]
*Alfa* $\longrightarrow$ `a` | `b` | ... | `z`
*Digit* $\longrightarrow$ `0` | `1` | ... | `9`
*Move* $\longrightarrow$ `F` | `B` | `J` | `L` | `R` | `O` | `I`

*Esempio 1.6.4* - Basandosi sulle regole definite sopra si possono generare le seguenti espressioni:

```
F
```

```
2[3FR]
```

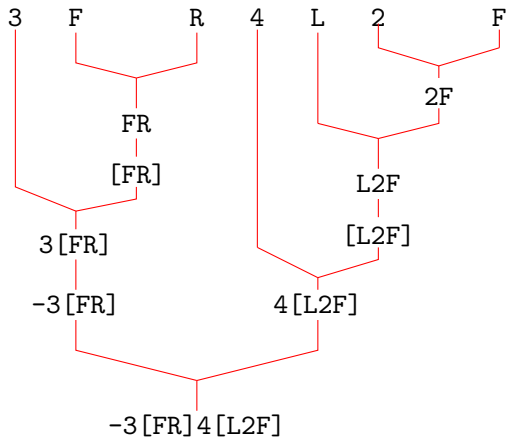
```
-4T[F2L]
```

```
---[RF-[FL]]3F
```

□

È possibile descrivere graficamente la generazione di un'espressione mediante un *albero sintattico* come si vede nell'esempio che segue.

*Esempio 1.6.5* - Il programma `-3[FR]4[L2F]` può essere costruito mediante un processo di generazione descritto dal seguente albero sintattico:




□

Chiameremo *espressione elementare* un movimento elementare `F`, `B`, `L`, `R`, `J`, oppure un'espressione della forma [...] oppure della forma $-\alpha$ dove $\alpha$ è un'espressione elementare.

## Trasformazioni e calcoli nell'algebra

Ogni ambiente algebrico è caratterizzato dall'esecuzione di procedimenti di trasformazione di espressioni in altre espressioni, in base a delle regole (definizioni, assiomi, teoremi, ...) specifiche dell'ambiente stesso. Tali procedimenti di trasformazione, detti anche *calcoli*, sono generalmente intenzionali e finalizzati alla riduzione delle espressioni verso forme equivalenti più semplici; si parla in questi casi di *semplificazioni*. Una processo di semplificazione reiterato fino a portare ad un singolo valore viene detto *valutazione*.

A seguire vedremo tutto ciò nel contesto dell'algebra LFR e descriveremo delle regole di trasformazione di un'espressione in una equivalente, definendo una sorta di calcolo algebrico delle espressioni. Useremo la notazione $\alpha \to \beta$ per indicare la trasformazione di un'espressione $\alpha$ in un'espressione $\beta$. Le regole di trasformazione sono generalmente legate ad una interpretazione delle espressioni in una contesto reale; in pratica una trasformazione $\alpha \to \beta$ indicherà che l'esecuzione di $\alpha$ avrà lo *stesso effetto* dell'esecuzione di $\beta$; in altre parole $\alpha$ e $\beta$ soddisfano ad uno specifico criterio di equivalenza. Per le espressioni dell'algebra LFR si possono distinguere diverse forme di equivalenza:

1. equivalenza di *scrittura*: quando le due espressioni sono descritte da stringhe uguali; ad esempio: `2FLRFR` e `2FLRFR`
2. equivalenza di *azioni*: quando le due espressioni descrivono la stessa sequenza di azioni, ossia la stessa traccia di esecuzione; ad esempio: `2[FFR]` e `2FR2FR`
3. equivalenza di *percorso*: quando l'esecuzione delle due espressioni fa compiere lo stesso percorso, portando allo stesso stato finale; ad esempio: `FR2F` e `2[FR]LF`

È facile dimostrare che le tre forme di equivalenza sopra definite soddisfano alle proprietà delle relazioni di equivalenza; inoltre, ciascuna forma di equivalenza è più stringente di quella che segue, nel senso che la prima implica la seconda e la seconda implica la terza. Ai fini delle trasformazioni delle espressioni ed aderentemente al contesto dei movimenti di un robot, quando a seguire useremo il termine *equivalente* riferito a delle espressioni intenderemo, senza altro avviso, l'equivalenza di percorso con raggiungimento dello stesso stato finale.

*Esempio 1.6.6 -* A seguire sono descritte due catene di trasformazioni (una di compressione e l'altra di espansione) basate sull'equivalenza di percorso:

$$\texttt{F2FR3F2R} \to \texttt{3FR3FRR} \to \texttt{2[3FR]R}$$
$$\texttt{2[L3FR]} \to \texttt{2[LFFFR]} \to \texttt{LFFFRLFFFR}$$

□

*Osservazione*. Una trasformazione della forma `2F` → `FF` è lecita se l'espressione `2F` non è preceduta da un moltiplicatore numerico; ad esempio non è lecita la catena di trasformazioni `3 2F` → `3FF` → `FFFF` → `4F`, mentre `3[2F]` è equivalente a `6F`.

Nelle trasformazioni $\alpha \to \beta$ cercheremo di applicare, per un principio di utilità, regole che *accorciano*, ossia la stringa $\beta$ dovrà risultare di lunghezza minore o uguale a quella di $\alpha$; questo comporterà che la complessità di $\beta$ risulti inferiore a quella di $\alpha$; in altri termini $\alpha \to \beta$ corrisponderà ad una semplificazione. La semplificazione si basa sulle seguenti regole di cancellazione e trasformazione, dove con $\epsilon$ si denota la *stringa vuota* costituita da nessun carattere.

$$\begin{aligned}
\texttt{FB} &\to \epsilon \\
\texttt{F-F} &\to \epsilon \\
\texttt{LR} &\to \epsilon \\
\texttt{RL} &\to \epsilon \\
k\texttt{R} &\to (k \bmod 4)\texttt{R} \\
k\texttt{L} &\to (k \bmod 4)\texttt{L} \\
3\texttt{R} &\to \texttt{L} \\
3\texttt{L} &\to \texttt{R}
\end{aligned}$$

Queste regole di semplificazione possono essere utilizzate unitamente alle proprietà delle operazioni descritte all'inizio di questo paragrafo, generando così dei processi di trasformazione che espandono o comprimono una stringa in una equivalente.

*Problema 1.6.1* Semplificare il programma `2[FR]2LF`.

*Soluzione*. Espandendo il programma in una stringa piatta ed elidendo la sottostringa `RL`, che risulta ininfluente, si ottiene:

$$\texttt{2[FR]2LF} \to \texttt{FRFRLLF} \to \texttt{FRFLF}$$

□

*Problema 1.6.2* Semplificare il programma `FFR-[BR]`.

*Soluzione*. Il programma può essere semplificato come segue:

$$\texttt{FFR-[BR]} \to \texttt{FFR-R-B} \to \texttt{FFRLF} \to \texttt{FFF} \to \texttt{3F}$$

□

## 1.7 L'algebra delle rotazioni

Consideriamo l'insieme delle rotazioni $\{\texttt{L}, \texttt{R}\}$ sul quale definiamo un'operazione binaria interna: dati due elementi generici $x$ ed $y$, denoteremo in notazione infissa con $x \circ y$, o semplicemente con $xy$, l'operazione fra $x$ ed $y$. Il risultato $z$ dell'operazione $x \circ y$ è definito come l'elemento dell'insieme delle rotazioni tale che lo stato finale che si ottiene eseguendo $z$ sia uguale a quello che si ottiene eseguendo $xy$; in altri termini il risultato di $xy$ dovrà corrispondere a "esegui la rotazione $x$ e poi la rotazione $y$". Si vede subito che `LR`, `RL`, `LL`, `RR` non corrispondono né a `L` nè a `R`. Per riuscire a valutare il risultato di questi casi, nell'interpretazione voluta, è necessario ampliare l'insieme $\{\texttt{L}, \texttt{R}\}$ con i seguenti due elementi:

- `O` : rotazione nulla (nessuna rotazione)
- `I` : inversione del verso di avanzamento (dietro-front)

Otteniamo così il seguente *insieme delle rotazioni*:

$$\mathcal{R} = \{\texttt{O}, \texttt{L}, \texttt{R}, \texttt{I}\}$$

Siamo ora in grado di scrivere la tabella dell'operazione $\circ$ sull'insieme $\mathcal{R}$:

| $\circ$ | O | L | R | I |
|---|---|---|---|---|
| O | O | L | R | I |
| L | L | I | O | R |
| R | R | O | I | L |
| I | I | R | L | O |

Dall'analisi della tabella si deduce che `O` costituisce l'elemento neutro dell'operazione $\circ$, ossia $x \circ \texttt{O} = \texttt{O} \circ x = x$ per ogni $x \in \mathcal{R}$. Inoltre, dalla simmetria della tabella rispetto alla diagonale di vertice $\circ$, si deduce che l'operazione $\circ$ è commutativa, ossia $x \circ y = y \circ x$ per ogni coppia di elementi $x$, $y$. L'operazione $\circ$ gode anche della proprietà associativa, ossia $(x \circ y) \circ z = x \circ (y \circ z)$ per ogni $x, y, z \in \mathcal{R}$. Tale proprietà può essere verificata direttamente analizzando tutte le $4^3$ diverse terne $x, y, z$, oppure ragionando in generale in termini di rotazioni, notando che una rotazione corrisponde ad eseguire l'addizione di 1 o $^-1$ al numero intero che rappresenta il verso corrente di avanzamento, sfruttando così l'associatività dell'operazione di addizione fra numeri interi. Disponiamo così di uno strumento che ci permette di valutare qualsiasi espressione formata da una sequenza di operandi dell'insieme delle rotazioni, eseguendo l'operazione fra coppie di operandi contigui, in base alla precedente tabella, iterando il procedimento fino ad arrivare ad un singolo operando il quale costituisce il risultato del processo di valutazione dell'espressione.

Le espressioni su $\mathcal{R}$ costituite da una sequenza di operandi (ad esempio `RRLILR`) possono essere generalizzate utilizzando tutte le possibilità del linguaggio `LFR` (concatenazione, moltiplicazione, raggruppamento, inversione), ottenendo così espressioni anche molto articolate (ad esempio `3RL4[RI-[2L]]`). Per queste espressioni si può applicare un processo di valutazione che genera un singolo valore dell'insieme delle rotazioni.

*Esempio 1.7.1* - Il processo di valutazione dell'espressione RRLOIRLLL può essere svolto a coppie di operandi, in parallelo; a seguire è descritto un esempio di calcolo:

$$\texttt{RRLOIRLLL} \to \texttt{(RR)(LO)(IR)(LL)L} \to \texttt{(IL)(LI)L} \to \texttt{(RR)L} \to \texttt{IL} \to \texttt{R}$$

oppure, sfruttando l'associatività a sinistra, valutare ad ogni passo l'operazione fra i primi due operandi dell'espressione ancora da valutare, fino ad arrivare ad un unico valore che costituisce il risultato finale del processo di valutazione:

$$\texttt{RRLOIRLLL} \to \texttt{(RR)(LOIRLLL)} \to \texttt{I(LOIRLLL)} \to \texttt{(IL)(OIRLLL)} \to \texttt{R(OIRLLL)} \to$$

$$\texttt{(RO)(IRLLL)} \to \texttt{R(IRLLL)} \to \texttt{(RI)(RLLL)} \to \texttt{L(RLLL)} \to \texttt{(LR)(LLL)} \to \texttt{O(LLL)} \to$$

$$\texttt{(OL)(LL)} \to \texttt{L(LL)} \to \texttt{(LL)L} \to \texttt{IL} \to \texttt{R}$$

□

*Esempio 1.7.2* - Basandosi sull'operazione di moltiplicazione di un numero naturale per un programma si può interpretare all'interno dell'algebra dell'operazione $\circ$ anche espressioni della forma 5R, come descritto nel seguente calcolo:

$$\texttt{5R} \to \texttt{RRRRR} \to \texttt{(RR)(RR)R} \to \texttt{(II)R} \to \texttt{OR} \to \texttt{R}$$

□

Nella valutazione delle espressioni, nello svolgimento dei calcoli parziali, oltre all'esecuzione diretta dell'operazione fra coppie di elementi (in base alla tabella dell'operazione $\circ$) si possono evitare alcuni calcoli, eseguendo delle riduzioni dell'espressione in base alle seguenti *regole di trasformazione*:

1. l'elemento neutro O può essere eliminato
2. le coppie LR e RL possono essere sostituite con O (e quindi eliminate in base alla regola 1.)
3. le stringhe LLL ed RRR possono essere sostituite, rispettivamente, dall'elemento R e dall'elemento L
4. le sequenze di operandi LL e RR possono essere sostituite con I
5. le sequenze di operandi LLLL e RRRR possono essere eliminate; in generale, si possono applicare le trasformazioni

   $$k\texttt{L} \to (k \bmod 4)\texttt{L}$$

   $$k\texttt{R} \to (k \bmod 4)\texttt{R}$$

6. la sequenza di operandi II può essere eliminata; in generale, si può applicare la trasformazione

   $$k\texttt{I} \to (k \bmod 2)\texttt{I}$$

7. un'espressione della forma [$x$] non preceduta da un moltiplicatore numerico né dal segno di inversione può essere sostituita con $x$

Queste regole di trasformazione vanno intese in senso bidirezionale anche se le trasformazioni inverse sono meno produttive e poco utilizzate in quanto tendono ad allungare la stringa.

*Esempio 1.7.3 -* Applicando le precedenti regole di trasformazione si possono svolgere i seguenti calcoli:

$$\texttt{LRL} \rightarrow \texttt{L}$$

$$\texttt{LLL} \rightarrow \texttt{R}$$

$$\texttt{5R} \rightarrow \texttt{R}$$

$$\texttt{[3RLR]} \rightarrow \texttt{3RLR} \rightarrow \texttt{3R} \rightarrow \texttt{L}$$

□

*Esempio 1.7.4 -* Le regole sopra descritte permettono di semplificare il processo di valutazione di un'espressione eseguendo dapprima delle semplificazioni formali dell'espressione da valutare ed eseguendo i calcoli dell'operazione $\circ$ solo alla fine. Ad esempio l'espressione dell'esempio 1.7.1 può essere semplificata e valutata come segue (sono sottolineate le parti dell'espressione che vengono trasformate):

$$\texttt{RRLOIRLLL} \rightarrow \texttt{R}\underline{\texttt{RL}}\,\underline{\texttt{O}}\texttt{IR}\underline{\texttt{LLL}} \rightarrow \underline{\texttt{RRRR}}\texttt{R} \rightarrow \texttt{R}$$

□

## 1.8 Compilazione ed ottimizzazione dei programmi

È interessante analizzare il meccanismo di funzionamento dell'automa Puntino in quanto riflette, benché in modo molto semplificato, l'architettura logica di esecuzione di molte macchine e computer attuali. Il *motore* di Puntino è in grado di eseguire *nativamente*, ossia senza alcun intervento di intermediazione, in sequenza le azioni elementari `F`, `B`, `L`, `R`, `J`, ossia, ad esempio, stringhe della forma `FFRFFLF`. Espressioni come `2[3FR]` o espressioni contenenti identificatori non sono invece comprese dal motore di Puntino ma richiedono una fase di preelaborazione che fornisca una equivalente stringa piatta direttamente interpretabile ed eseguibile da Puntino. Questa fase di trasformazione viene detta *compilazione*, l'espressione iniziale (`2[3FR]`) viene detta *codice sorgente* e la stringa ottenuta (`FFFRFFFR`) viene detta *codice macchina* [5]. A seguire la fase di compilazione viene solitamente svolta la fase di *ottimizzazione* che consiste nella semplificazione del codice macchina prodotto dalla fase di compilazione eliminando le parti che producono un effetto neutro e nel sostituire porzioni di codice con altre equivalenti ma più efficienti in modo da ridurre la complessità del programma. La fase di compilazione genera un'espressione equivalente alla sorgente, mentre la fase di ottimizzazione produce un'espressione ottimizzata equivalente a quella compilata. Ad esempio, la sottostringa `LR` può essere eliminata, la sottostringa `RRRRR` può essere semplificata in `R` e la sottostringa

[5] Nel caso di Puntino il linguaggio macchina è un sottoinsieme del linguaggio di programmazione `LFR`; per la maggior parte degli elaboratori il linguaggio macchina è un linguaggio a più basso livello, distinto dal linguaggio di programmazione con cui è scritto il codice sorgente.

RRR può essere sostituita con una sola L. Queste fasi di preelaborazione dal linguaggio dell'utente (linguaggio di programmazione) al linguaggio dell'esecutore (linguaggio macchina) caratterizza la quasi totalità delle macchine. La situazione è descritta nella figura 1.2 [6] .

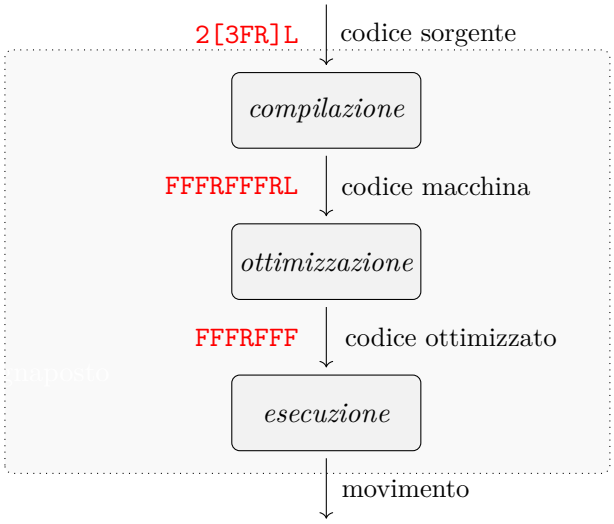



Figura 1.2: Schema di compilazione, ottimizzazione ed esecuzione di un programma: dal codice sorgente creato dal solutore al codice macchina ottimizzato eseguito dall'esecutore.

*Esempio 1.8.1* - Il programma 3R2[L3FR] viene trasformato mediante le fasi della compilazione ed ottimizzazione come segue (sono sottolineate le parti di stringa che vengono trasformate nella fase di ottimizzazione):

$$\texttt{3R2[L3FR]} \rightarrow \underline{\texttt{RRR}}\texttt{LFFF}\underline{\texttt{RL}}\texttt{FFFR} \rightarrow \texttt{LLFFFFFFR}$$

□

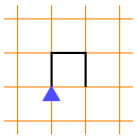
*Osservazione*. In fase di ottimizzazione una stringa come FB non può essere eliminata in quanto l'effetto della sua esecuzione genera un effetto collaterale costituito dal tratto di linea che viene disegnato.

[6]Sono solitamente predisposte due modalità di traduzione: una, detta propriamente *compilazione*, che consiste nella traduzione dell'intero codice sorgente in codice macchina prima dell'esecuzione; un'altra, detta *interpretazione*, che traduce il codice sorgente linea per linea durante l'esecuzione.

## 1.9 Equazioni e sistemi di equazioni

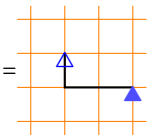
Un tipico problema dell'algebra consiste nel determinare il valore di una o più incognite data un'equazione o un sistema di equazioni. Tale problema può essere trasportato anche nel contesto dell'automa Puntino prendendo come incognite dei comandi (definiti mediante dei sottoprogrammi in cui codice costituisce proprio il valore delle incognite da determinare) e come equazioni delle figure che vengono disegnate da combinazioni di questi comandi. Per limitare il possibile numero (addirittura infinito) delle soluzioni, si assumono le seguenti convenzioni: che la soluzione sia ridotta ai minimi termini che il percorso non ripassi su tratti già disegnati e che venga definito il punto di partenza. Ulteriori informazioni possono essere date fornendo la posizione ed il verso di avanzamento all'inizio ed alla fine della figura. A seguire indicheremo con $\phi(\alpha)$ la figura disegnata dall'esecuzione del programma $\alpha$; con ▲ e △ verranno indicate le posizioni ed i versi di Puntino all'inizio ed alla fine del disegno della figura.

*Esempio 1.9.1* - Data l'equazione

$\phi(3x) =$ 

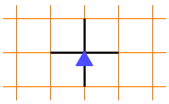
si deduce facilmente che, con le convenzioni precedentemente indicate, $x$ = `FR`. □

*Esempio 1.9.2* - Data l'equazione

$\phi(2x\mathtt{F}) =$ 

ragionando un po', si deduce che $x$ = `LFR`. □

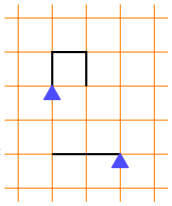
*Problema 1.9.1* Risolvere l'equazione

$\phi(4x) =$ 

*Soluzione*. La soluzione è $x$ = `F2RJR`. Basandosi sull'asimmetria che non esiste un comando analogo a `J` che salta *indietro*, questa soluzione può essere migliorata con $x$ = `BJR`. □

*Problema 1.9.2* Risolvere il seguente sistema di equazioni (si assuma l'ipotesi che non sia lecito passare su un tratto già disegnato):

$$\begin{cases} \phi(3x) = \\ \phi(2[xy]) = \end{cases}$$



[1]

[2]

*Soluzione*. Si tratta di un tipico problema di ricerca la cui soluzione richiede delle prove organizzate all'interno di una strategia di ricerca. Dall'analisi dell'equazione [1] si deduce che l'incognita $x$ deve contenere un singolo comando di movimento (`F` o `B`) seguito o preceduto da una rotazione (`L` o `R`); $x$ deve pertanto essere una delle seguenti espressioni: $x$ = `FR`, $x$ = `RF`, $x$ = `BL`, $x$ = `LB`. Dal fatto che la figura dell'equazione [2] contenga solo due tratti si deduce che $y$ è composta da zero, una o più delle seguenti azioni:

- rotazione a sinistra (`L`)
- rotazione a destra (`R`)
- salto in avanti (`J`)

La soluzione è pertanto la seguente: $x$ = `FR`, $y$ = `L`. □

*Osservazione*. L'attività di ricerca del significato dei comandi assomiglia alla soluzione di un cruciverba in cui l'avanzamento della scoperta delle parole che vengono scritte agevola sempre più la ricerca delle successive parole mancanti, magari con un feedback retroattivo per l'eventuale correzione di precedenti parole già scritte.

## 1.10 Una vecchia tartaruga per disegnare

La maestra Ivana ha trovato nella soffitta della scuola un vecchio e strano macchinario, dalle fattezze di una tartaruga, risalente probabilmente ai primi anni settanta del secolo scorso. Dal fatto che sotto all'apparecchiatura, in zona centrale, ci fosse un alloggiamento per una penna, la maestra ha dedotto che doveva trattarsi di una "tartaruga per disegnare" su un foglio di carta. La conferma è derivata dall'osservazione che sui 4 tasti della console che comanda la tartaruga c'erano delle frecce (↑, ↓, ↶, ↷) corrispondenti ai 4 possibili movimenti elementari della tartaruga.

Figura 1.3: Una foto degli anni sessanta, con una tartaruga simile a quella trovata dalla maestra.

Dopo aver acceso le apparecchiature, la maestra ha provato a pigiare, un po' a caso, sui tasti della console ed ha notato che la tartaruga si muoveva similmente a Puntino, lungo le linee di un ideale piano quadrettato. Ma, forse per qualche malfunzionamento, la tartaruga si comportava in modo inusuale e non immediatamente decifrabile. Per capirne il comportamento la maestra ha provato a digitare in sequenza i tasti con l'intenzione di disegnare un quadrato di lato unitario, corrispondentemente al programma $4[\uparrow\curvearrowright]$, ossia, nel linguaggio `LFR`, al programma `4[FR]`. Eseguendo questo programma la tartaruga ha eseguito alcune rotazioni sul posto, senza disegnare alcuna linea. La maestra, con l'intenzione di disegnare un quadrato di lato 2 unità, ha eseguito un programma equivalente a $4[2\uparrow\curvearrowleft]$ ottenendo il disegno di un tratto unitario con riposizionamento della tartaruga allo stato iniziale. Ragionando su questo strano comportamento la maestra ha dedotto, giustamente, che erano stati scambiati alcuni tasti della console, inavvertitamente o da parte di qualche burlone.

*Problema 1.10.1* In base agli effetti dell'esecuzione dei programmi $4[\uparrow\curvearrowright]$ e $4[2\uparrow\curvearrowleft]$ sopra descritti, determinare la corretta permutazione dei 4 tasti della console in modo che la tartaruga esegua correttamente i movimenti coerentemente con il segno presente su ciascun tasto.

*Soluzione*. Dal fatto che $4[\uparrow\curvearrowright]$ non comporti alcuno spostamento si deduce che né $\uparrow$ né $\curvearrowright$ sono azioni di movimento corrispondenti a `F` o `B`, ma solo rotazioni `L` o `R`. Dal fatto che $4[2\uparrow\curvearrowleft]$ disegna un segmento unitario si deduce che $2\uparrow\curvearrowleft$ disegna un segmento unitario e riporta allo stato di partenza. Dal confronto fra queste due situazioni si deduce che $\uparrow$=`L`, $\curvearrowleft$=`R`, $\curvearrowright$=`B`; di conseguenza $\downarrow$=`F`. □

## ESERCIZI

**1.1** Stabilire se le seguenti stringhe sono espressioni corrette secondo la sintassi del linguaggio `LFR`. Determinare la traccia delle espressioni sintatticamente corrette.

1. `-2FR`
2. `-[2FR]`
3. `-[2[FR]]`
4. `2[-FR]`

**1.2** Determinare la traccia delle azioni elementari generate dall'esecuzione delle seguenti espressioni:

1. `3FR2B`
2. `3[2FR]`
3. `3[2[FR]LF]`
4. `-[F-2[BR]]`

**1.3** Determinare la linea del percorso generato dalle seguenti espressioni:

1. `4[2FR]`
2. `2L3[FRFL]`
3. `4[FL-JF]`
4. `2[3[FR]L]`

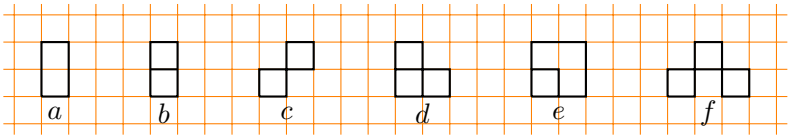
**1.4** Partendo dallo stato iniziale $(0, 0, \texttt{NORTH})$, stabilire lo stato finale raggiunto al termine dell'esecuzione di ciascuna delle seguenti espressioni:

1. `3[2F3L]`
2. `3[F2[3FR]]`
3. `2FR-[F2R]`
4. `2[-B-[3[F-L]]]`

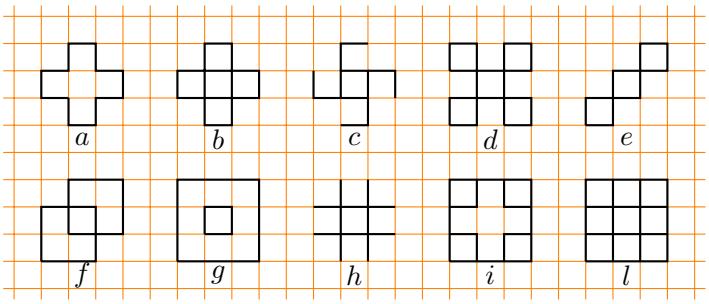
**1.5** Descrivere l'effetto e valutare la complessità dei seguenti programmi:

1. `3FRF2L`
2. `2[3FR3[FL]2L]`

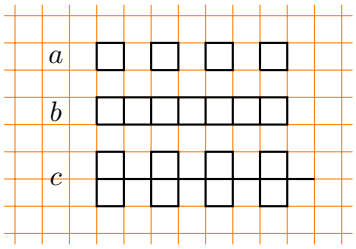
**1.6** Disegnare le seguenti figure, scegliendo il punto di partenza più adatto, evitando di passare su un tratto già disegnato e senza usare il comando `J` di salto:



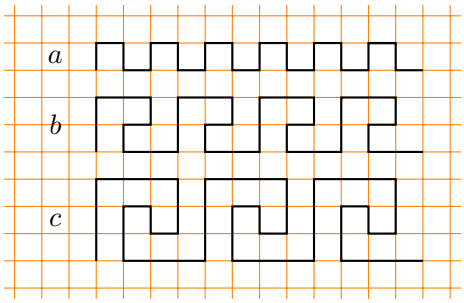
1.7 Disegnare le seguenti figure, scegliendo il punto di partenza più adatto, evitando di passare su un tratto già disegnato ed usando il comando J di salto solo se necessario:




1.8 Disegnare le seguenti figure mediante un'unica spezzata, senza ripassare su tratti già disegnati.




1.9 Disegnare le seguenti figure, individuando un motivo grafico che si ripete. In base alla seguenza delle 3 figure descritte individuare uno schema generale di una famiglia di figure ad onda caratterizzate da un *ordine di increspatura* (le figure presentate corrispondono agli ordini 1, 2 e 3). In particolare, descrivere e disegnare le figure corrispondenti agli ordini 0 e 4.




1.10 Sia $\alpha$ una generica espressione del linguaggio LFR. Discutere l'effetto dell'espressione $\alpha - \alpha$.

**1.11** Sia $M$ un movimento di $\mathcal{M} = \{\texttt{F}, \texttt{B}, \texttt{L}, \texttt{R}\}$ ed $m, n$ due generici numeri naturali. Dimostrare che valgono le seguenti proprietà:

1. $mM\ nM \equiv (m+n)M$
2. $mM - nM \equiv (m-n)M$
3. $m[nM] \equiv (mn)M$

**1.12** Scrivere un algoritmo per semplificare una stringa di soli caratteri `L` ed `R`.

**1.13** Descrivere mediante degli alberi sintattici la generazione delle seguenti espressioni:

1. `FLFRRFFL`
2. `[FF][L[FR]]`
3. `3F4[FR]2L`

**1.14** Espandere ed ottimizzare le seguenti espressioni:

1. `3R2[R2F]`
2. `4[L2F3R]2L`
3. `R3[LFR]2R4[FLR]`

**1.15** Valutare le seguenti espressioni di rotazioni:

1. `3R2LIR`
2. `2[3RL]3[2IL]`
3. `3[R2[RIL]]`

**1.16** Stabilire se le seguenti coppie di espressioni sono fra loro equivalenti:

1. `LR` e `RL`
2. `FR` e `RF`
3. `FB` e `BF`
4. `LRL` e `RLR`

**1.17** Determinare tutti gli insiemi minimali (di minima cardinalità) $\mathcal{M}_0$ di movimenti potenzialmente equivalenti all'insieme dei movimenti $\mathcal{M} = \{\texttt{F}, \texttt{B}, \texttt{L}, \texttt{R}\}$. Esprimere ciascun movimento di $\mathcal{M} \setminus \mathcal{M}_0$ come combinazione dei movimenti di $\mathcal{M}_0$.

**1.18** Stabilire la relazione che esiste fra una figura e la figura che si ottiene scambiando fra loro i comandi `L` ed `R`. E scambiando fra loro i comandi `F` e `B` ? Ed eseguendo entrambi gli scambi?

**1.19** Si supponga di poter utilizzare solo i comandi `F` ed `R`. Assegnare alle variabili *left* e *back* delle espressioni che risultino equivalenti rispettivamente ad `L` e `B`.

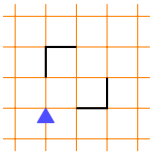
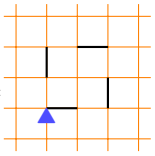
**1.20** Avendo eseguito l'assegnazione $x$=`3R`, definire in funzione di $x$ le variabili *sinistra*, *destra*, *inverti* di evidenti significati.

**1.21** Determinare l'ambiente finale che si ottiene come effetto dell'esecuzione della seguente sequenza di assegnazioni:

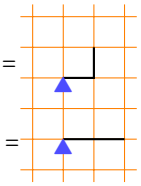
```
x = yRz
y = xF
z = 2FR
```

Descrivere come si modifica l'ambiente aggiungendo come quarta la (apparentemente ininfluente) assegnazione `x=x`.

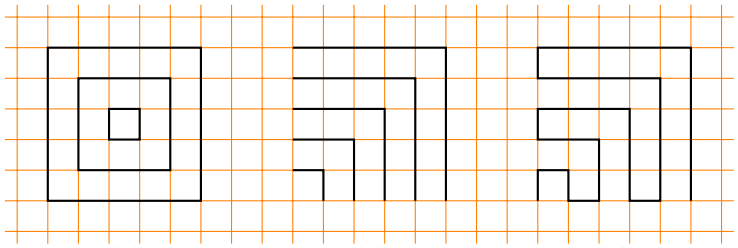
**1.22** Risolvere le seguenti due equazioni:

$\phi(2x) =$  $\phi(4x) =$ 

**1.23** Risolvere il seguente sistema di equazioni:

$$\begin{cases} \phi(2x) = \text{  } & [1] \\ \phi(xyx) = & [2] \end{cases}$$

**1.24** Determinare le espressioni per disegnare le figure descritte a seguire:



$(a)$ $(b)$ $(c)$

# 2

# INTERAGIRE

*È notevole che nella storia della scienza quasi tutti i fenomeni siano stati presto o tardi spiegati in termini di interazione fra parti prese a due a due.*

Marvin Minsky, *La società della mente*

I procedimenti di movimento diventano particolarmente interessanti se l'entità che si muove è in grado di rapportarsi con l'ambiente circostante e con altre entità dell'ambiente. Questa capacità viene generalmente definita come *sentire* e viene espletata attraverso i sensi. Calandosi nel contesto della robotica, la facoltà di sentire viene realizzata mediante i *sensori*. Un'ulteriore capacità consiste nell'esplicare delle azioni sull'ambiente esterno, mediante componenti del robot dette genericamente *attuatori*. Il ricorso duplice e sinergico alle azioni di sentire ed agire viene globalmente detto *interagire*.

L'uso di sensori ed attuatori richiede un cambio di marcia nella programmazione del robot in quanto insorge l'esigenza di gestire delle *condizioni* che descrivono il rapporto del robot con gli oggetti presenti nell'ambiente esterno. Questo richiede l'uso ad un linguaggio più evoluto; a questo scopo il linguaggio `LFR` viene esteso mediante i controlli i cosidetti *controlli della programmazione strutturata*.

In questo capitolo si vira verso una situazione più "realistica": l'automa Puntino assume una sua fisicità spaziale e diventa un quadretto, dotato di sensori, che si muove sul piano quadrettato; per meglio sperimentare con i sensori il suo movimento viene confinato all'interno di una scacchiera di cui, in ogni istante, occupa esattamente una casella.

## 2.1 Muoversi su una scacchiera

Consideriamo una porzione limitata di piano, strutturata come una scacchiera di $8 \times 8$ caselle, ciascuna individuata da una coppia di numeri interi, secondo il metodo delle coordinate cartesiane. In questo spazio si muove un robot elementare che in ogni istante occupa una casella, è in grado di avanzare di una casella alla volta, ha una direzione e verso di avanzamento (indicheremo con `NORTH`, `EAST`, `SOUTH`, `WEST` i versi corrispondenti ai quattro punti cardinali) ed è in grado di ruotare, a sinistra ed a destra, di un angolo retto. Chiamiamo questo robot con il termine *Quadretto*.

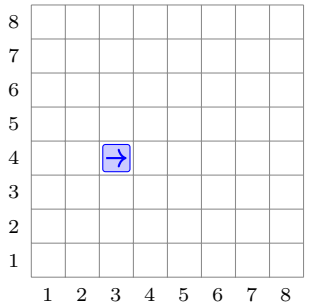



Figura 2.1: Il robot *Quadretto* nel suo ambiente. La freccia denota il verso di avanzamento.

Il movimento, nella situazione semplificata di spazio discretizzato che stiamo considerando, può essere ottenuto combinando pochissime operazioni elementari: avanzamento ed indietreggiamento di una casella e rotazione a sinistra o a destra di un angolo retto; queste azioni elementari possono essere concatenate, organizzate in sequenza e mediante ripetizione. In ogni istante la situazione di Quadretto è completamente descritta dal suo *stato*, costituito dalla *posizione* in cui si trova e dal *verso* di avanzamento. Lo stato risulta dunque descrivibile mediante una terna della forma (*riga*, *colonna*, *verso*) formata dalle coordinate di posizione e dal verso di avanzamento. Ad esempio $(3, 4, \texttt{EAST})$ denota lo stato di Quadretto quando è posto nella casella di coordinate $(3, 4)$ e rivolto verso est (come descritto nella figura 2.1). Chiameremo *stato iniziale* lo stato $(1, 1, \texttt{NORTH})$, ossia lo stato di Quadretto posto nell'angolo in basso a sinistra e rivolto verso nord.

*Osservazione*. Il fatto che Quadretto si sposti in modo regolare e preciso nelle caselle semplifica la sua gestione, permettendo di concentrarsi sulla logica del movimento, filtrando tutti quegli aspetti pratici, approssimativi ed imponderabili che si incontrano nella gestione di un robot reale.

Quadretto è comandabile mediante un insieme di comandi elementari che permettono di avanzare di 1 passo (passare alla casella davanti), di indietreggiare di 1 passo (passare alla casella dietro) e di ruotare a sinistra o destra di un angolo retto, rimanendo sulla stessa casella. Secondo il paradigma *strumento*, Quadretto può essere movimentato utilizzando i tradizionali tasti freccette presenti sulla tastiera; questa modalità permette all'utente, usando un programma di simulazione, di immedesimarsi in Quadretto, di sperimentare ed esercitarsi sul concetto di destra/sinistra e di rinforzare il senso di orientamento nel piano. Più interessante risulta l'adozione del paradigma *solutore-esecutore* che consiste nella scrittura di un programma per Quadretto utilizzando un adeguato linguaggio di programmazione.

Per muovere Quadretto utilizzeremo il linguaggio `LFR` già utilizzato per l'automa Puntino, al cap. *Muovere*. Per Quadretto le azioni elementari hanno i seguenti effetti:

- `F` : avanzamento alla casella successiva
- `B` : indietreggiamento alla casella precedente
- `L` : rotazione a sinistra di un angolo retto (rimanendo sulla casella)
- `R` : rotazione a destra di un angolo retto (rimanendo sulla casella)

*Osservazione*. Quando ci si muove in uno spazio finito (scacchiera) bisogna stabilire come si reagisce quando si arriva ad una situazione limite. Per Quadretto adotteremo la convenzione più semplice: quando arriva al limite (contatto con il bordo della scacchiera) esso rimane fermo anche se sollecitato ad avanzare [1].

*Esempio 2.1.1* - L'esecuzione del programma `2FL2F` fa compiere un percorso ad angolo retto verso sinistra, con i lati di lunghezza pari a 3 caselle. □

*Esempio 2.1.2* - L'esecuzione del programma

```
3[2FRF]
```

comporta l'esecuzione della seguente sequenza di azioni elementari:

```
FFRFFFRFFFRF
```

□

*Problema 2.1.1* Determinare lo stato finale dopo aver eseguito il seguente programma a partire dallo stato iniziale $(3, 2, \texttt{EAST})$:

```
4[2F3[RF]]
```

*Soluzione*. Simulando l'esecuzione del programma passo-dopo-passo (a mente, con carta e penna, su una scacchiera oppure al computer) si ricava che alla fine si raggiunge lo stato $(3, 2, \texttt{EAST})$. □

[1] Non è comunque questa l'unica scelta: si potrebbe, ad esempio, invertire il senso di marcia oppure continuare su un altro lato della scacchiera.

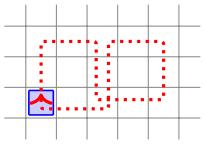
## 2.2 Problemi di movimento

Molti problemi di movimento hanno la seguente struttura: "stabilire l'effetto di un dato programma" oppure "scrivere un programma che fa compiere un dato percorso o raggiungere un dato stato finale". Anche i problemi della forma "scrivere un programma per eseguire una transizione di stato $s_0 \to s_f$" si risolvono individuando dapprima un adeguato percorso, riconducendosi quindi al caso precedente. Un percorso si presta naturalmente ad essere scomposto in sotto percorsi, secondo gli schemi che ricalcano proprio le strutture per la costruzione dei programmi. E tutti questi schemi di scomposizione/composizione di percorsi possono avvalersi delle metodologie top-down e bottom-up.

*Problema 2.2.1* Stabilire l'effetto dell'esecuzione del seguente programma, nell'ipotesi che Quadretto non incontri ostacoli lungo il percorso:

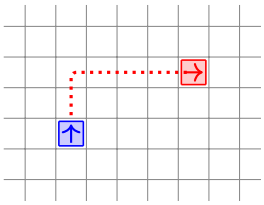
```
3[2FR]2L4[2FL]2R2FR
```

*Soluzione*. Analizzando le singole componenti del programma si deduce che la sua esecuzione ha l'effetto di muovere Quadretto su un percorso ad "otto", con ritorno allo stato di partenza, come descritto dalla traiettoria punteggiata della figura che segue.



□

*Problema 2.2.2* Muovere Quadretto lungo un percorso ad angolo come descritto nella seguente figura:



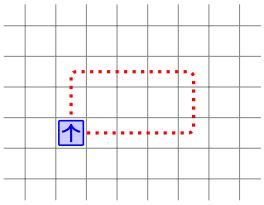
*Soluzione*. Applicando la metodologia top-down, il percorso può essere scomposto in due tratti lineari raccordati da una rotazione a destra, come descritto nell'algoritmo 1:

**Algoritmo 1** - percorso ad angolo

| | |
|---|---|
| 1: avanza di 2 passi | ▷ `2F` |
| 2: ruota a destra | ▷ `R` |
| 3: avanza di 4 passi | ▷ `4F` |

che corrisponde al programma `2FR4F`. □

*Problema 2.2.3* Muovere Quadretto lungo un percorso rettangolare come descritto nella seguente figura:



*Soluzione*. Applicando la metodologia bottom-up, il percorso può essere scomposto in due tratti ad angolo raccordati da una rotazione a destra, come descritto nell'algoritmo 2:

**Algoritmo 2** - percorso rettangolare

| | |
|---|---|
| 1: **for** 2 **times** | |
| 2: percorso dell'esempio 2.2.2 | ▷ `2FR4F` |
| 3: ruota a destra | ▷ `R` |
| 4: **end for** | |

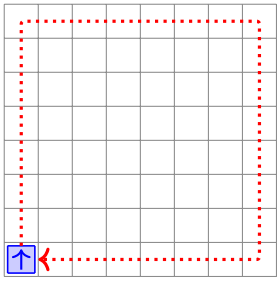
che corrisponde al programma `2[2FR4FR]`. □

Nell'affrontare la soluzione dei problemi un buon solutore dovrebbe sempre cercare di individuare degli schemi generali che possano applicarsi a categorie di problemi similari. In questo modo, una volta individuata la categoria di problemi di appartenenza, la soluzione di un problema risulta di molto agevolata. Anche per i problemi di movimento accade spesso che un movimento articolato risulti composto da una ripetizione per un numero $k$ di volte (*frequenza*) di un movimento base $m$ (*periodo*). In questi casi la soluzione del problema si riduce all'individuazione del numero $k$ e del movimento base $m$, riconducendosi al seguente schema algoritmico:

**Algoritmo 3** - movimento di frequenza $k$ e periodo $m$

1: **for** $k$ **times**
2: esegui movimento $m$
3: **end for**

*Problema 2.2.4* Quadretto si trova nello stato iniziale $(1, 1, \texttt{NORTH})$. Muovere Quadretto lungo un percorso ciclico quadrato attorno al bordo interno della scacchiera $8 \times 8$, ritornando al punto di partenza, come descritto nella figura che segue.

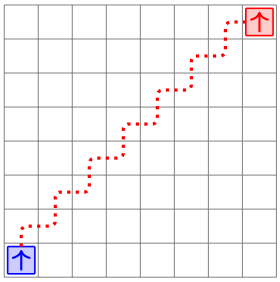


*Soluzione*. Il percorso complessivo è composto da 4 movimenti lungo i lati, raccordati fra loro da una rotazione a destra. Ciascuno di questi movimenti è composto da 7 azioni semplici di avanzamento (`7F`). Pertanto la soluzione del problema è data dal programma

```
4[7FR]
```

Si noti che lo schema algoritmico corrisponde all'applicazione della metodologia top-down che dapprima individua il sottoproblema "percorrere un lato della scacchiera" e successivamente questo viene scomposto in "azioni elementari di avanzamento (`F`)". □

*Problema 2.2.5* Muovere Quadretto dallo stato iniziale $(1, 1, \texttt{NORTH})$ allo stato finale $(8, 8, \texttt{NORTH})$ in modo da passare su tutte le caselle della diagonale principale $(1, 1) - (8, 8)$, come descritto nella figura che segue:



*Soluzione*. In questo caso il movimento elementare consiste nel fare un "gradino" mediante l'espressione

```
FRFL
```

Ripetendo 7 volte questo movimento elementare si risolve il problema mediante il programma

```
7 [FRFL]
```

□

## 2.3 Problemi di calcolo

Quadretto può essere utilizzato come un rudimentale calcolatore numerico in grado di eseguire le operazioni aritmetiche fondamentali. A tale scopo bisogna innanzitutto superare lo scoglio dato dal fatto che Quadretto "si muove" ma "non calcola". Questo disallineamento fra la modalità operativa del robot (movimento sulla scacchiera) e l'obiettivo da raggiungere (risultato del calcolo) impone di individuare preliminarmente dei meccanismi per specificare i dati in ingresso sui quali eseguire i calcoli ed un meccanismo per ricevere i risultati prodotti dal procedimento di calcolo. Osserviamo che Quadretto è un automa che, eseguendo un programma, compie una transizione da uno stato iniziale ad un stato finale, dove ciascuno stato è composto da una posizione sulla scacchiera e da un verso di avanzamento. Da questa caratterizzazione minimale si può ricavare una prima strategia generale di come calcolare: basta interpretare lo stato iniziale come 'portatore' dei dati (numeri) sui quali eseguire i calcoli e lo stato finale come 'indicatore' del risultato finale. In particolare, le coordinate della posizione iniziale forniscono una coppia di valori naturali che possono essere assunti come dati sui quali eseguire le operazioni aritmetiche binarie (addizione, sottrazione, moltiplicazione, divisione, ...). Un modo alternativo, che sarà adottato a seguire, per specificare i dati consiste nell'inserirli nel programma che viene eseguito, ad esempio specificandoli direttamente nelle espressioni del programma o definendo delle costanti o, ancora, mediante delle istruzioni di lettura dei dati dell'utente).

Vediamo ora come realizzare alcune semplici operazioni. Le operazioni aritmetiche di addizione, sottrazione e moltiplicazione sui numeri naturali sono strettamente connaturate alla struttura del linguaggio `LFR` e si prestano ad essere facilmente realizzate adottando la seguente strategia di calcolo: i numeri sui quali eseguire le operazioni vengono specificati nel programma `LFR`; il calcolo viene svolto mediante l'esecuzione del programma; il risultato viene fornito in base alla casella sulla quale si ferma Quadretto al termine dell'esecuzione del programma. Per rendere più facilmente interpretabile il risultato è sufficiente predisporre la scacchiera sulla quale si muove Quadretto scrivendo in ciascuna cella un numero che precisa il risultato.

Per le operazioni di addizione, sottrazione e moltiplicazione è sufficiente utilizzare una scacchiera unidimensionale costituita da un sufficiente (in base alla grandezza degli operandi) numero di colonne e da una sola riga, con le caselle numerate come illustrato nella figura 2.2. Inizialmente Quadretto viene posizionato sulla prima casella (numero 0) ed orientato verso est (ad esempio eseguendo il programma `HR`) [2] .

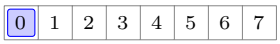



Figura 2.2: Il robot *Quadretto* sulla scacchiera monodimensionale, adatta per eseguire le operazioni di addizione, sottrazione e moltiplicazione.

I seguenti esempi descrivono le operazioni di addizione, sottrazione e moltiplicazione.

*Esempio 2.3.1 -* (*Addizione*) L'operazione $m + n$ di addizione fra due numeri naturali è realizzata mediante il programma

$$m\mathtt{F}n\mathtt{F}$$

L'idea è semplice: partendo dalla casella iniziale denotata con il numero 0, avanzando di $m$ caselle si arriva sulla casella $m$ e con altri $n$ avanzamenti si arriva sulla casella $m + n$. Al termine dell'esecuzione Quadretto sarà fermo sulla casella denotata dal numero $m + n$. □

*Esempio 2.3.2 -* (*Sottrazione*) Similmente al precedente esempio relativo all'addizione, l'operazione $m - n$ di sottrazione fra due numeri naturali, nell'ipotesi che sia $m \geq n$, può essere realizzata mediante il programma

$$m\mathtt{F}n\mathtt{B}$$

Avanzando di $m$ caselle e successivamente indietreggiando di $n$ caselle, al termine dell'esecuzione Quadretto sarà fermo sulla casella denotata dal numero $m-n$. Nel caso in cui sia $n \geq m$ si otterrà 0 come risultato, in quanto Quadretto si fermerà sulla prima casella contrassegnata dal numero 0. □

[2] Si può utilizzare anche una scacchiera con più di una riga, usandone solo la prima.

*Esempio 2.3.3* - (*Moltiplicazione*) L'operazione $m * n$ di moltiplicazione fra due numeri naturali può essere ottenuta ripetendo $m$ volte $n$ avanzamenti di casella; ciò corrisponde all'esecuzione del programma

$$mn\mathbf{F}$$

□

Altre operazioni quali la divisione intera non sono realizzabili mediante la parte del linguaggio `LFR` fin qui descritta, ma richiedono altre caratteristiche del linguaggio che saranno esaminate più avanti ed algoritmi di calcolo più articolati.

## 2.4 Oggetti sulla scacchiera

Sulle caselle della scacchiera in cui si muove Quadretto si possono posizionare oggetti di due tipologie:

- *muri*: oggetti fissi che impediscono il movimento
- *blocchi*: oggetti che possono essere spinti e spostati

I muri si comportano come la parte esterna alla scacchiera e vincolano il movimento di Quadretto; i blocchi, invece, sono degli oggetti che possono essere spinti da Quadretto quando sono a suo diretto contatto; la spinta causata dall'avanzamento di Quadretto ha l'effetto di spostare alla casella di fronte il blocco a contatto, a patto che sia libera la casella che dovrebbe essere occupata dal blocco che viene spinto in avanti, o comunque sia libera la casella (contrassegnata con ∘ nella figura 2.3) alla fine della fila di blocchi posta a diretto contatto con Quadretto.

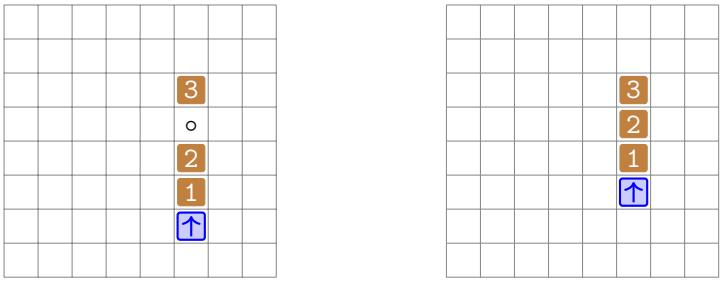



Figura 2.3: Situazione prima e dopo la spinta di Quadretto.

La spinta può avvenire anche con un indietreggiamento. Un blocco posto in un angolo della scacchiera non può essere spostato; un blocco posto a contatto con il bordo può essere spinto lungo il bordo della scacchiera ma non può essere staccato dal contatto con il bordo.

Sulla scacchiera popolata da muri e blocchi si possono verificare delle situazioni di *stallo* qualora sia inibito il movimento di Quadretto; queste situazioni corrispondono al caso in cui Quadretto sia a contatto diretto con un muro oppure con un blocco che non può essere spinto.

## 2.5 I sensori

La gestione del movimento in un ambiente in cui sono presenti oggetti richiede che il robot sia dotato di *sensori* che lo informano su alcune caratteristiche dell'ambiente circostante e gli permettono di rapportarsi ed interagire con gli oggetti dell'ambiente. Un sensore è caratterizzato dalla diversa tipologia di informazione esterna che è in grado di recepire.

Una primitiva forma di interazione con l'ambiente circostante consiste nell'avere consapevolezza della propria collocazione rispetto all'ambiente stesso. Ciò è reso possibile dalle seguenti tipologie di *sensori di stato*:

- sensore di *posizione*: analogo ad un GPS degli attuali dispositivi mobili, informa il robot della sua posizione rispetto ad un prefissato sistema di riferimento solidale allo spazio esterno
- sensore di *orientamento*: analogo ad una bussola, fornisce al robot il suo verso di avanzamento rispetto ad una direzione di riferimento

Un'altra classe di sensori permette di ricevere informazioni di varia natura relative alla presenza di oggetti presenti nell'ambiente:

- sensore di *spazio*: rileva se davanti al robot c'è spazio per avanzare
- sensore di *contatto*: sente quando il robot ha la sua parte frontale in contatto con un oggetto
- sensore di *tatto*: riesce a qualificare la diversa tipologia dell'oggetto con il quale è in contatto
- sensore di *pressione*: sente se davanti c'è un oggetto che non può essere spinto
- sensore di *distanza*: fornisce la distanza dall'oggetto posto davanti nella direzione di avanzamento del robot
- sensore di *colore*: percepisce il colore dell'oggetto davanti

In un contesto di programmazione i sensori possono essere distinti in

- sensori *numerici* che ritornano un valore numerico
- sensori *logici* che corrispondono ad una condizione

## 2.6 I sensori di Quadretto

I sensori di Quadretto, sia quelli numerici che logici, sono assimilabili ad una funzione che ritorna un valore numerico, adottando la tradizionale convenzione che 0 denoti `FALSE` ed 1 `TRUE`. Questa convenzione consente di inserire efficacemente i sensori nel contesto del linguaggio `LFR`, usandoli come moltiplicatori di espressioni. A seguire sono descritte le varie tipologie di sensori di Quadretto.

### Sensori di stato

I sensori di stato di Quadretto sono realizzati mediante le seguenti funzioni:

- `X` : valore da 1 a 8 corrispondente alla coordinata orizzontale della casella in cui è posizionato Quadretto
- `Y` : valore da 1 a 8 corrispondente alla coordinata verticale della casella in cui è posizionato Quadretto
- `A` : angolo di avanzamento attuale, ossia uno dei quattro valori 0 (`NORTH`), 1 (`WEST`), 2 (`SOUTH`), 3 (`EAST`)

In un contesto di programmazione i sensori sopra descritti equivalgono a delle funzioni che ritornano dei valori naturali che dipendono dalla particolare situazione in cui si trova Quadretto nel momento in cui queste funzioni vengono richiamate. I valori ritornati da queste funzioni possono essere utilizzati come moltiplicatori delle espressioni di movimento.

*Esempio 2.6.1* - Grazie agli specifici valori formiti dal sensore `A`, per orientare Quadretto verso nord è sufficiente eseguire il programma

```
AR
```

□

### Sensore di distanza

Il *sensore di distanza* rileva la distanza dall'oggetto davanti, fornendo il numero di caselle di separazione dall'oggetto. È gestito dalla seguente funzione:

- `D` : distanza (in caselle) dall'oggetto davanti (*Distance*)

*Esempio 2.6.2* - Il seguente programma avanza Quadretto fino ad arrivare a contatto con un oggetto:

```
DF
```

Operativamente, viene valutato il valore che indica la distanza dal più vicino oggetto posto davanti a Quadretto e poi avanza di un numero di caselle pari a questo valore. □

## Sensore di colore

Un *sensore di colore* fornisce un'informazione sul colore della casella davanti. Nella situazione minimalista di Quadretto, tale sensore distingue solamente una casistica bicolore. Viene gestito mediante la seguente funzione:

`C` : 0 se la casella davanti a Quadretto è di colore nero; 1 in tutti gli altri casi (*Color*)

*Esempio 2.6.3* - Per ruotare Quadretto fino a trovare una casella frontale o laterale nera, si può usare il seguente programma

```
L3[CR]
```

□

## Sensore di spazio

Il *sensore di spazio* rileva se la casella davanti al robot è libera da oggetti. Viene gestito mediante la seguente funzione:

`S` : 1 se la casella davanti è libera; 0 se la casella davanti è occupata da un oggetto (*Space*)

*Esempio 2.6.4* - Per avanzare Quadretto di al più 2 caselle, fermandosi prima se si arriva a contatto con un oggetto, si può usare il seguente programma

```
2[SF]
```

□

## Sensore di tocco

Un *sensore di tocco* rileva il contatto diretto con un oggetto. Viene gestito mediante la seguente funzione:

`T` : 1 se Quadretto è a contatto con un oggetto davanti; 0 se la casella davanti è libera (*Touch*)

*Esempio 2.6.5* - Per ruotare Quadretto al fine di trovare una strada libera da ostacoli per avanzare si può ricorrere al seguente programma:

```
4[TR]
```

□

## Sensore di pressione

Il *sensore di pressione* permette di riconoscere se l'oggetto davanti oppone resistenza all'avanzamento; nel caso di Quadretto si attiva quando si verifica una situazione di stallo, ossia quando l'ultimo tentativo di avanzamento non è riuscito a causa di un muro o di un blocco che non può essere spinto a causa

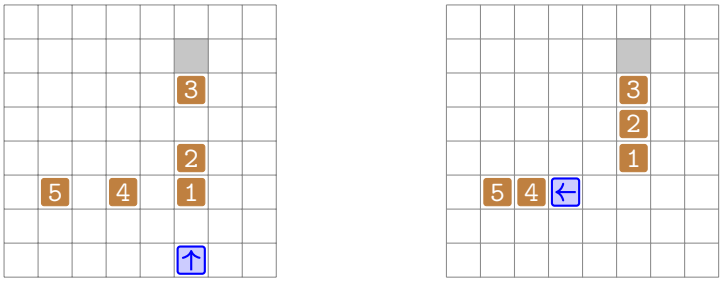
della mancanza di spazio. Il sensore di pressione viene gestito mediante la seguente funzione:

`P` : 1 se l'ultimo tentativo di movimento non è riuscito a causa di una situazione di impedimento dovuto ad un blocco da parte di un oggetto; 0 altrimenti (*Pressure*)

*Esempio 2.6.6* - Nella figura che segue è illustrato l'effetto dell'esecuzione del programma

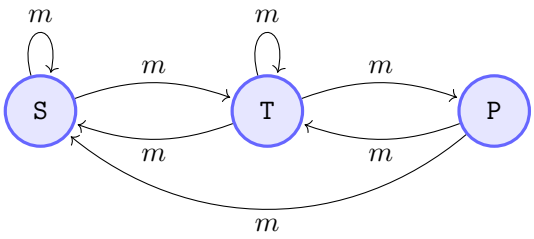
```
5[FPL]
```

in una specifica configurazione della scacchiera in cui sono presenti 5 blocchi ed 1 muro.




□

*Osservazione*. I sensori di spazio, tocco e pressione costituiscono una partizione di tutti i possibili stati in cui può trovarsi Quadretto in relazione agli oggetti presenti nella casella davanti. La situazione può essere descritta mediante il seguente grafo di transizione di stato:




In questo grafo con $m$ viene denotato un generico movimento elementare (`F`,`B`,`L`,`R`); gli stati sono denotati dalla corrispondente funzione; in pratica nello stato Ⓢ significa che è vera la condizione `S` (`S`=1) (ed analogamente per gli altri stati). Notiamo che le transizioni di stato dipendono anche dall'ambiente, in particolare dall'oggetto che si trova nella casella davanti a Quadretto dopo aver eseguito il movimento $m$. Questo fatto comporta che gli stati del grafo abbiano due uscite.

*Osservazione*. Mediante i sensori di tocco (`T`) e pressione (`P`) non è possibile stabilire il tipo di oggetto (muro o blocco) posto nella casella davanti.

A seguire vengono presentati alcuni problemi che richiedono l'uso dei sensori precedentemente descritti; per la loro soluzione viene applicata la metodologia bottom-up, sfruttando le soluzioni presentate nei precedenti esempi, e la metodologia top-down di scomposizione in sottoproblemi.

*Problema 2.6.1* Orientare Quadretto verso Nord ed avanzare fino ad arrivare a contatto frontale con un oggetto.

*Soluzione*. Componendo le soluzioni dei due esempi 2.6.2 e 2.6.1 si arriva alla seguente soluzione:

```
ARDF
```

□

*Problema 2.6.2* Ruotare Quadretto al fine di trovare una strada per avanzare e, se trovata, avanzare di al più 3 caselle.

*Soluzione*. La soluzione è fornita combinando in sequenza le due soluzioni dei problemi visti negli esempi 2.6.5 e 2.6.4, ottenendo il programma

```
4[TR]3[SF]
```

□

Un algoritmo, in particolare quando si adotta un approccio top-down di scomposizione in sottoproblemi, viene generalmente espresso in termini discorsivi che successivamente vengono codificati in un linguaggio di programmazione. Nella stesura del programma l'intelaiatura discorsiva dell'algoritmo può essere mantenuta come commento al codice, come si vede nel problema che segue.

*Problema 2.6.3* Portare Quadretto alla posizione $(8, 8)$ e rivolto verso Nord.

*Soluzione*. Una soluzione è fornita dal seguente algoritmo dove, sulla destra, è riportata la codifica in linguaggio `LFR`.

| | |
|---|---|
| 1: orientamento verso Nord | ▷ `AR` |
| 2: avanzamento fino al muro | ▷ `DF` |
| 3: rotazione a destra | ▷ `R` |
| 4: avanzamento fino al muro | ▷ `DF` |
| 5: rotazione a sinistra | ▷ `L` |

Riunendo le soluzioni dei sotto problemi si ottiene la codifica in linguaggio `LFR` mediante la stringa `ARDFRDFL`. La codifica può essere resa più intellegibile se

viene suddivisa in linee, ciascuna corrispondente ad un singolo sotto problema; per aumentare la leggibilità si possono riportare come commenti esplicativi, alla destra del codice, le linee dell'algoritmo suddiviso in più linee.

```
AR   # orientamento verso Nord
DF   # avanzamento fino al muro
R    # rotazione a destra
DF   # avanzamento fino al muro
L    # rotazione a sinistra
```

□

*Problema 2.6.4* Portare Quadretto allo stato iniziale $(1, 1, \texttt{NORTH})$.

*Soluzione*. A seguire è riportato un algoritmo che risolve il problema; a lato è riportata la codifica in linguaggio `LFR` di ciascuna istruzione.

**Algoritmo 4** - stato iniziale $(1, 1, \texttt{NORTH})$

| | |
|---|---|
| 1: orientamento a nord | ▷ `AR` |
| 2: indietreggiamento alla prima riga | ▷ `YB` |
| 3: orientamento a est | ▷ `R` |
| 4: indietreggiamento alla prima colonna | ▷ `XB` |
| 5: orientamento a nord | ▷ `L` |

Componendo in sequenza le soluzioni di ciascun sottoproblema si ottiene il programma complessivo costituito dalla stringa `ARYBRXBL`. □

## 2.7 La memoria di Quadretto

Per ogni macchina (automa, robot, computer, ...) si possono distinguere due macro tipologie di memoria: la *memoria interna* che rappresenta il ricordo individuale di alcuni aspetti della vita trascorsa e la *memoria esterna* registrata esternamente all'entità. Anche per Quadretto ritroviamo queste due tipologie di memoria, descritte nelle seguenti sotto sezioni.

### La pila delle azioni

Tutte le azioni di movimento `F`, `B`, `L`, `R` eseguite da Quadretto che comportano un effettivo cambiamento di stato (sono quindi escluse tutte le `F` e `B` che in fase di esecuzione hanno trovato impedimento in un muro o in un oggetto bloccato) vengono memorizzate in una *memoria a pila* gestibile mediante i seguenti comandi e funzioni:

- `N` : comando che *inizializza* il contenuto della pila vuota (`N`ew)
- `M` : funzione che ritorna la *sequenza delle azioni* presenti nella pila (`M`emory)

La pila delle azioni eseguite consente di tentare dei percorsi e poi ritornare al punto di partenza.

*Esempio 2.7.1 -* Il programma

```
7F-M
```

avanza Quadretto fino a toccare (o sbattere contro) il muro davanti e poi lo riporta, in retromarcia, al punto di partenza. □

### La memoria della scacchiera

Un modo alternativo per memorizzare delle informazioni consiste nello scrivere su un supporto esterno ed avere la possibilità di accedere a quanto scritto. Per Quadretto il supporto esterno è costituito dalla scacchiera in cui si muove; lo scrivere consiste nel marcare una casella con un segno convenzionale. La lettura avviene mediante un sensore che ritorna 1 se la casella è marcata. La memoria della scacchiera è gestita mediante i seguenti comandi e funzioni:

- `W` : comando che *marca* la casella corrente (`W`rite)
- `U` : comando che *cancella* la marcatura della casella corrente (`U`nmark)
- `Q` : funzione d'*interrogazione* che ritorna 1 se la casella corrente è marcata, 0 altrimenti (`Q`uery)

Inizialmente tutte le caselle della scacchiera non sono marcate.

*Esempio 2.7.2 -* Per marcare le 4 caselle in contatto a Quadretto sui 4 lati di Quadretto si può eseguire il seguente programma:

```
4[FWBL]
```

□

## 2.8 Programmazione strutturata

Parallelamente all'evoluzione dei linguaggi di programmazione verso forme di più alto livello che man mano si sono allontanate dalle architetture fisiche delle macchine e dalle modalità operative delle stesse, si sono sviluppate delle tecniche di programmazione con l'obiettivo di agevolare il solutore nello sviluppo degli algoritmi. Fra queste tecniche ha avuto particolare successo la *programmazione strutturata* sviluppatasi con l'obiettivo di costruire algoritmi combinando pochi e ben ben formati schemi di base al fine di migliorare la realizzazione, la leggibilità e la manutenibilità dei programmi.

Secondo le indicazioni della programmazione strutturata e sulla base di alcuni risultati della teoria della programmazione (teorema di Böhm-Jacopini), ogni algoritmo può essere costruito utilizzando solo tre schemi fondamentali, *sequenziale*, *condizionale* e *ciclico*. Questi controlli sono disponibili anche nel linguaggio `LFR` come descritto nelle seguenti sotto sezioni.

### Controllo sequenziale

Il controllo *sequenziale* è la forma più elementare di strutturazione delle azioni: le azioni sono descritte in sequenza, una dopo l'altra, e vengono eseguite in sequenza nell'ordine indicato, secondo il seguente schema algoritmico:

1: $\alpha_1$
2: $\alpha_2$
3: ...
4: $\alpha_n$

*Esempio 2.8.1* - Vogliamo portare Quadretto nell'angolo situato davanti alla propria destra. Il problema si presta naturalmente ad una scomposizione sequenziale in sotto movimenti, come descritto nel seguente algoritmo (alla destra di ciascuna linea è riportata la codifica in linguaggio `LFR`).

| | |
|---|---|
| 1: *avanza fino al muro* | ▷ `DF` |
| 2: *ruota a destra* | ▷ `R` |
| 3: *avanza fino al muro* | ▷ `DF` |

Ricomponendo i tre movimenti riportati nell'algoritmo si ottiene l'espressione complessiva

`DFRDF`

□

## Controllo condizionale

Nel controllo *condizionale* le azioni sono rette da condizioni che specificano se eseguirle o meno. Lo schema base è il seguente:

```
1: if C then
2:     α
3: end if
```

che indica che l'espressione $\alpha$ deve essere eseguita solo se è verificata la condizione $\mathcal{C}$.

Nel linguaggio `LFR` i sensori `S`, `T`, `P` costituiscono un rudimentale meccanismo per realizzare delle condizioni logiche a basso livello, adottando la tradizionale interpretazione del valore 0 come `FALSE` ed il valore 1 come `TRUE`. Per realizzare il controllo condizionale viene usato il valore fornito dal sensore come moltiplicatore di un'espressione di movimento. Indicando con $\mathcal{C}$ la condizione corrispondente ad un sensore di Quadretto, il precedente schema condizionale viene tradotto nel linguaggio `LFR` con

$$\mathcal{C}\,\alpha$$

*Esempio 2.8.2 -* Vogliamo avanzare Quadretto di una casella, solo se c'è spazio davanti. Dal punto di vista logico si tratta di realizzare la seguente porzione di algoritmo:

```
1: if c'è spazio davanti then
2:     avanza
3: end if
```

Nel linguaggio `LFR` questa porzione di algoritmo può essere espressa semplicemente moltiplicando l'espressione da eseguire in modo prefisso per il valore fornito dal sensore corrispondente alla condizione *c'è spazio davanti*; in pratica:

```
SF
```

□

*Esempio 2.8.3 -* Vogliamo avanzare Quadretto di una casella, invertendo il verso di avanzamento nel caso in cui si trovi a contatto con un oggetto. Si tratta di realizzare la seguente porzione di algoritmo:

```
1: if c'è spazio davanti then
2:     avanza
3: else
4:     inverti verso
5:     avanza
6: end if
```

In questo caso non è immediatamente applicabile l'idea di soluzione vista nel precedente esempio in quanto siamo in presenza di un controllo condizionale a due rami. Ad una semplice analisi si nota però che la precedente porzione di algoritmo è equivalente alla seguente:

```
1: if non c'è spazio davanti then
2:     inverti verso
3: end if
4: avanza
```

Quest'ultima trasformazione porta immediatamente alla soluzione

`T2RF`

□

Non sempre uno schema condizionale a due rami risulta convertibile in uno ad un solo ramo, come visto nel precedente esempio. In talune situazioni serve ricorrere allo schema condizionale a due rami:

```
1: if C then
2:     α
3: else
4:     β
5: end if
```

che indica di eseguire l'espressione $\alpha$ se la condizione $\mathcal{C}$ è verificata, altrimenti l'espressione $\beta$. Nel linguaggio `LFR` questo schema viene tradotto nella forma

$$\mathcal{C}\,\alpha : \beta$$

Nel caso in cui l'espressione $\alpha$ sia ridotta all'espressione nulla, il precedente schema condizionale a due rami si riduce al seguente:

```
1: if ¬C then
2:     β
3: end if
```

che trova la sua corrispondente codifica nel linguaggio LFR con

$$\mathcal{C} : \beta$$

*Esempio 2.8.4 -* Vogliamo far eseguire a Quadretto un singolo movimento effettivo (avanzamento o rotazione), un avanzamento se non è a contatto con ostacoli, altrimenti una rotazione a destra. Ciò corrisponde alla seguente porzione di algoritmo:

```
1: if c'è spazio davanti then
2:     avanza
3: else
4:     destra
5: end if
```

che, in base a quanto visto appena sopra, viene tradotto mediante la seguente espressione:

$$\texttt{SF:R}$$

□

*Problema 2.8.1* Spiegare perchè il programma SFTR non traduce correttamente la porzione di algoritmo presentata nell'esempio 2.8.4.

*Soluzione*. Nel caso in cui Quadretto parta da una condizione di distanza di 1 casella dall'oggetto davanti, in quanto, dopo aver eseguito F, diventa vera la condizione T che comporta l'esecuzione anche della rotazione finale R. □

## Controlli ciclici

I controlli ciclici consentono di descrivere l'esecuzione ripetuta di azioni. Di questi controlli esistono diverse forme che si differenziano per la modalità di terminazione del ciclo.

Il *controllo ciclico predefinito* permette di eseguire un'espressione per un numero predefinito $k$ di volte. Viene descritto dallo schema

```
1: for k times
2:     α
3: end for
```

che indica di ripetere $k$ volte l'espressione $\alpha$. Nel linguaggio LFR viene scritto con la sintassi

$$k\,[\alpha]$$

*Esempio 2.8.5 -* Il seguente algoritmo marca le 4 caselle adiacenti a Quadretto:

```
1: for 4 times
2:     avanza
3:     marca
4:     indietro
5:     destra
6: end for
```

Nel linguaggio `LFR` questo algoritmo si esprime con `4[FWBR]`. □

Il *controllo ciclico indefinito* permette di eseguire un'espressione indefinitamente. Viene descritto dallo schema

```
1: loop
2:     α
3: end loop
```

che indica di ripetere indefinitamente le azioni $\alpha$. Nel linguaggio `LFR` viene scritto con la sintassi

$$\{\alpha\}$$

*Esempio 2.8.6 -* L'esecuzione del programma `{2FR}` muove Quadretto indefinitamente lungo un quadrato di lato 3 caselle, in senso orario. □

Generalmente un ciclo termina al verificarsi o meno di specifiche condizioni che vengono valutate ad ogni esecuzione del ciclo. Una forma particolarmente interessante e frequente, detta *controllo ciclico condizionato in testa*, permette di eseguire un'azione fintantoché una data condizione è verificata; tale controllo viene descritto mediante lo schema algoritmico

```
1: while C do
2:     α
3: end while
```

che indica di ripetere le azioni $\alpha$ fintantoché la condizione $\mathcal{C}$ è verificata. Un'altra, detta *controllo ciclico condizionato in coda*, si esprime secondo lo schema algoritmico:

```
1: repeat
2:     α
3: until C
```

che indica di ripetere le azioni $\alpha$ fino al verificarsi della condizione $\mathcal{C}$.

Per realizzare queste forme di cicli, nel linguaggio `LFR` è previsto il seguente comando:

`K` : *interrompe l'esecuzione* delle azioni interne al ciclo (più interno) in cui si trova il comando `K` (breaK)

Usando questo comando il controllo **while** $\mathcal{C}$ **do** $\alpha$ **end while** viene codificato con

$$\{\mathcal{C}\texttt{:K}\ \alpha\}$$

mentre lo schema **repeat** $\alpha$ **until** $\mathcal{C}$ viene codificato con

$$\{\alpha\ \mathcal{C}\ \texttt{K}\}$$

*Esempio 2.8.7* - Vogliamo avanzare Quadretto fino a raggiungere una situazione di contatto frontale con un oggetto. Dal punto di vista logico si tratta di realizzare la seguente porzione di algoritmo:

```
1: while c'è spazio davanti do
2:     avanza
3: end while
```

che nel linguaggio `LFR` viene tradotto con una delle seguenti equivalenti forme:

`{SF:K}` `{S:KF}` `{TKF}` `{TK:F}`

Osserviamo che, nonostante l'apparente equivalenza, il programma `{S:KF}` è diverso dal programma `DF` nei casi in cui, in fase di esecuzione, venga tolto o portato un oggetto sul percorso davanti a Quadretto. □

*Problema 2.8.2* Portare Quadretto a contatto con il bordo e farlo girare indefinitamente lungo il bordo. Si assuma l'ipotesi che non sano presenti oggetti sulla scacchiera.

*Soluzione*. Si tratta di ripetere l'azione "*portati sul bordo e ruota a destra*". Questa azione risolve sia il sottoproblema iniziale di portarsi a contatto con il bordo che quello di percorre un lato del bordo nelle successive rotazioni. La soluzione dell'intero problema può essere formulata con un ciclo indefinito `{ }` esterno che ingloba un ciclo finito `{SF:K}`, ossia

`{{SF:K}R}`

□

*Problema 2.8.3* Muovere indefinitamente Quadretto, evitando eventuali oggetti presenti sulla scacchiera.

*Soluzione*. La soluzione può essere basata sulla ripetizione ciclica di un avanzamento di un passo; nel caso Quadretto si trovi a contatto con un oggetto, prima di avanzarlo di un passo è sufficiente farlo ruotare in modo da trovare una strada libera per avanzare; a tale scopo è sufficiente eseguire il seguente ciclo di rotazioni: `{SKR}`; pertanto per fare un passo di avanzamento è sufficiente eseguire la seguente espressione: `{SKR}F`. La soluzione completa del problema è fornita dunque dalla seguente espressione:

```
{{SKR}F}
```

□

## Operatori logici nel linguaggio `LFR`

Nel linguaggio `LFR` non sono previsti i tradizionali operatori logici *and* ($\wedge$), *or* ($\vee$) e not ($\neg$). Qualora le espressioni logiche contenenti questi operatori siano contestualizzate nei controlli condizionale e ciclico è possibile tradurre gli operatori logici come descritto nella tabella che segue dove sono indicati a sinistra i controlli condizionali e ciclici secondo la notazione algoritmica ed a destra la corrispondente codifica nel linguaggio `LFR`. Con $\mathcal{C}$, $\mathcal{C}_1$ e $\mathcal{C}_2$ sono denotate delle condizioni logiche elementari corrispondenti ai sensori di Quadretto.

| Notazione algoritmica | Codifica nel linguaggio `LFR` |
|---|---|
| **if** $\mathcal{C}$ **then** $\alpha$ **end if** | $\mathcal{C}\,\alpha$ |
| **if** $\mathcal{C}$ **then** $\alpha$ **else** $\beta$ **end if** | $\mathcal{C}\,\alpha : \beta$ |
| **if** $\neg\mathcal{C}$ **then** $\alpha$ **end if** | $\mathcal{C} : \alpha$ |
| **if** $\mathcal{C}_1 \wedge \mathcal{C}_2$ **then** $\alpha$ **end if** | $\mathcal{C}_1\mathcal{C}_2\,\alpha$ |
| **if** $\mathcal{C}_1 \vee \mathcal{C}_2$ **then** $\alpha$ **end if** | $\mathcal{C}_1\,\alpha : [\mathcal{C}_2\,\alpha]$ |
| **if** $\mathcal{C}_1 \wedge \mathcal{C}_2$ **then** $\alpha$ **else** $\beta$ **end if** | $\mathcal{C}_1\,[\mathcal{C}_2\,\alpha] : \beta$ |
| **if** $\mathcal{C}_1 \vee \mathcal{C}_2$ **then** $\alpha$ **else** $\beta$ **end if** | $\mathcal{C}_1\,\alpha : [\mathcal{C}_2\,\alpha] : \beta$ |
| **while** $\mathcal{C}$ **do** $\alpha$ **end while** | $\{\mathcal{C}\,\texttt{:K}\ \alpha\}$ |
| **while** $\neg\mathcal{C}$ **do** $\alpha$ **end while** | $\{\mathcal{C}\,\texttt{K}\ \alpha\}$ |
| **while** $\mathcal{C}_1 \wedge \mathcal{C}_2$ **do** $\alpha$ **end while** | $\{\mathcal{C}_1\,\texttt{:K}\ \mathcal{C}_2\,\texttt{:K}\ \alpha\}$ |
| **while** $\mathcal{C}_1 \vee \mathcal{C}_2$ **do** $\alpha$ **end while** | $\{\mathcal{C}_1\,\texttt{:}\ \mathcal{C}_2\,\texttt{:K}\ \alpha\}$ |

*Osservazione*. Con riferimento alla tabella precedente, nelle due situazioni corrispondenti ai controlli che iniziano con **if** $\mathcal{C}_1 \vee \mathcal{C}_2$ **then** ... bisogna ripetere la scrittura dell'espressione $\alpha$; nei casi in cui l'espressione $\alpha$ sia complessa, per facilitare la scrittura, conviene ricorrere ad una preventivaa assegnazione della forma "$a = \alpha$" per poter scrivere l'identificatore $a$ al posto dell'espressione $\alpha$.

## 2.9 La programmazione mediante sensori

I sensori logici corrispondono ad una condizione. Nel caso di Quadretto i sensori logici di colore (`C`), spazio (`S`), tocco (`T`), pressione (`P`) possono essere usati come moltiplicatori nelle espressioni; visto poi i particolari valori che possono assumere (0 o 1) possono essere impiegati per esprimere delle condizioni. Se $C$ è un generico sensore logico corrispondente ad una condizione $\mathcal{C}$ ed $\alpha$ è una generica azione, la scrittura $C\ \alpha$ corrisponde all'espressione "se è vera la condizione $\mathcal{C}$ allora esegui l'azione $\alpha$".

*Esempio 2.9.1* - L'espressione "*se tocchi allora gira a sinistra*" può essere codificata con l'espressione `TL`. □

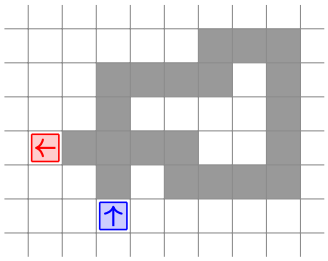
I sensori di Quadretto sono stati denotati mediante una singola lettera maiuscola (`D`, `S`, ...); in aggiunta, per ciascuno di essi, è stato indicato un nome esteso (*Distance*, *Space*, ...) che suggerisce il significato del sensore stesso. In fase di lettura di una porzione di codice `LFR` si può utilizzare questo nome, ai fini di una maggiore intellegibilità. Similmente i comandi di movimento (`F`, `B`, `L`, `R`) possono essere letti come *forward*, *backward*, *left* e *right*. Nel caso di sensori logici risulta significativo far precedere la locuzione "if" mentre ai sensori numerici risulta utile precedere la locuzione "for" ed aggiungere alla fine "endfor". Ad esempio, la stringa `ALS[FTR]` può essere letta, in modo più rilassato, "*for A times left endfor if space then forward if touch then right endif endif*" che, se scritta secondo le convenzioni degli algoritmi strutturati, avrebbe una forma ancora più leggibile:

```
1: for A times
2:     left
3: end for
4: if space then
5:     forward
6:     if touch then
7:         right
8:     end if
9: end if
```

Spesso il processo è inverso: si esprime prima l'algoritmo in modo strutturato, usando i controlli della programmazione strutturata, e poi si codifica nel linguaggio di programmazione desiderato, come si vede nel prossimo paragrafo.

## 2.10 Seguire una linea

Un classico problema per robot che si muovono su un piano consiste nel percorrere una linea, fino ad arrivare all'altra estremità della linea. Si suppone che il robot abbia un sensore di colore frontale per orientarsi lungo la linea. Nel caso di Quadretto utilizziamo il sensore di colore e supponiamo che la linea sia di colore `BLACK` e sia garantita la precondizione di contatto frontale con l'inizio della linea da percorrere; il percorso termina quando Quadretto esce dalla linea. Una situazione, riferita a Quadretto, è descritta nella figura che segue.



Si tratta di un problema semplice ma di soluzione non immediata; risulta quindi poco indicato procedere subito alla codifica in linguaggio `LFR` ma bisogna dapprima individuare un algoritmo risolutivo e solo successivamente passare alla codifica. La strategia per percorrere la linea è descritta nell'algoritmo 5.

**Algoritmo 5** - *segui linea*

```
1: while c'è linea davanti do
2:     avanza
3:     cerca linea
4: end while
```

Indicando con *color* il sensore di colore della casella davanti a Quadretto e con `BLACK` il colore della linea da seguire, la condizione 1 : *c'è linea davanti* si esprime con

$$color = \texttt{BLACK}$$

mentre l'azione 3 : *cerca linea*, ha lo scopo di orientare Quadretto in modo da guadagnare la condizione di avere la linea nera davanti a sè, e si esprime mediante la seguente porzione di algoritmo:

```
1: if color ≠ BLACK then
2:    sinistra
3:    if color ≠ BLACK then
4:       inverti
5:       if color ≠ BLACK then
6:          sinistra
7:       end if
8:    end if
9: end if
```

Passiamo ora alla codifica in linguaggio LFR. L'algoritmo 5 viene codificato con

{CKF $P$ }

dove con $P$ è indicata la codifica del sottoproblema 'cerca linea'. Seguendo pedissequamente la struttura dell'algoritmo che risolve il sottoproblema si arriva alla sua codifica che è:

C[LC[2R]CL]

Innestando questa porzione di codice nella precedente si ottiene la codifica complessiva:

{CKFC[LC[2R]CL]}

## 2.11 I sottoprogrammi

Un potente strumento di programmazione è costituito dai *sottoprogrammi*, ossia dalla possibilità di definizione di porzioni di programma ed alla loro identificazione mediante un nome con il quale può essere richiamato in vari punti del programma. Dal punto di vista operativo la definizione di un sottoprogramma costituisce un comodo artificio per condensare in una parola una parte articolata di programma, con una evidente economia di scrittura (specialmente se il programma viene richiamato più volte), favorendo la leggibilità (per il solutore) del programma stesso; dal punto di vista metodologico risulta lo strumento principale a sostegno delle metodologie top-down e bottom-up; dal punto di vista dell'interazione fra solutore e esecutore la definizione di un sottoprogramma rappresenta la possibilità offerta al solutore di istruire l'esecutore, aumentando il repertorio delle capacità di base dell'esecutore stesso.

La tecnica programmativa dei sottoprogrammi si fonda su due momenti logicamente e temporalmene distinti: la *definizione* e la *chiamata*. La definizione consiste nell'assegnare un nome identificativo ad una porzione di programma mentre la chiamata consiste nello scrivere il nome del sottoprogramma nel punto in cui si vuole che venga chiamato. La definizione permette solo di specificare le azioni da eseguire, mentre, affinché le azioni vengano svolte si deve richiamare il sottoprogramma scrivendo come istruzione il nome identificativo con il quale il sottoprogramma è stato definito. Al di là del tecnicismo sottostante alla modalità di esecuzione di una chiamata ad un sottoprogramma, è logicamente coerente pensare che al posto dell'identificatore venga sostituita (ed eseguita) l'espressione che lo definisce, oppure, equivalentemente, pensare che il controllo dell'esecuzione passi all'espressione che definisce il sottoprogramma e che al termine dell'esecuzione di questa espressione il controllo ritorni all'esecuzione dell'azione immediatamente successiva all'identificatore.

Per definire un sottoprogramma useremo la notazione

$$id \stackrel{\text{def}}{=} exp$$

dove *id* è l'identificatore del sottoprogramma ed *exp* è l'espressione che viene assegnata all'identificatore *id*, costituita da un generico programma; nel punto della chiamata del sottoprogramma viene scritto l'identificatore *id* e viene eseguita l'espressione *exp* ottenendo il corrispondente effetto.

*Osservazione*. La sintassi della definizione dei sottoprogrammi come descritto sopra è allineata alle possibilità sintattiche del linguaggio `LFR` ed è molto limitata rispetto a quanto previsto nei linguaggi di programmazione più evoluti; in particolare non è possibile usare dei parametri; ciò sarà possibile al capitolo *Esplorare* dove verrà utilizzato il linguaggio Python.

## Assegnazioni e definizioni

Al di là di una apparente analogia sintattica fra assegnazioni e sottoprogrammi, questi ultimi si differenziano dalle assegnazioni e si caratterizzano per i seguenti aspetti:

- un sottoprogramma può essere definito una sola volta e non può trovarsi all'interno di altre espressioni
- l'espressione che compare alla destra in una definizione di un sottoprogramma non viene valutata al momento della definizione ma viene memorizzata com'è; viene eseguita al momento della chiamata
- quando in un programma si incontra l'identificatore di un sottoprogramma si recupera l'espressione corrispondente alla sua definizione e si esegue tale espressione
- se nella definizione di un sottoprogramma l'identificatore del sottoprogramma compare sulla destra, al momento della prima chiamata del sottoprogramma si innesca una chiamata ricorsiva; ciò consente di realizzare dei cicli mediante dei sottoprogrammi
- nell'espressione che compare sulla destra di una definizione di un sottoprogramma può comparire solo l'identificatore del sottoprogramma stesso (sottoprogramma ricorsivo) o altri identificatori di sottoprogrammi, ma non identificatori di variabile (questo affinché il sottoprogramma sia svincolato ed indipendente dall'ambiente delle variabili)
- se in un'espressione che viene assegnata ad un identificatore compare l'identificatore di un sottoprogramma esso non viene sostituito nella valutazione dell'espressione ma verrà eseguito al momento dell'esecuzione dell'espressione corrispondente alla variabile

*Esempio 2.11.1 -* La scrittura

$$inverti \stackrel{\text{def}}{=} \texttt{2L}$$

definisce un sottoprogramma di nome *inverti*. Con il comando *inverti* il sottoprogramma viene eseguito e si ottiene l'effetto di invertire il senso di marcia di Quadretto. □

*Esempio 2.11.2 -* L'esecuzione del sottoprogramma

$$duepassi \stackrel{\text{def}}{=} \texttt{2F}$$

ha l'effetto di far avanzare Quadretto, nella direzione corrente, di 2 passi. □

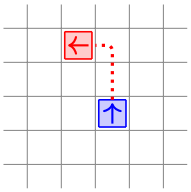
Un sottoprogramma può essere richiamato all'interno della definizione di un altro sottoprogramma. È questa una tipica situazione che si verifica quando si fa ricorso alle metodologie top-down e bottom-up.

*Esempio 2.11.3* - Per far fare 2 passi indietro a Quadretto si può definire e successivamente richiamare il seguente sottoprogramma che richiama i due sottoprogrammi *inverti* e *duepassi* definiti nei due esempi precedenti:

$$indietroduepassi \stackrel{\text{def}}{=} [inverti \;\; duepassi \;\; inverti]$$

□

*Problema 2.11.1* Muovere Quadretto a mossa di cavallo degli scacchi, come descritto dalla figura che segue.



*Soluzione*. L'immediata soluzione è data dal programma `2FLF`. Per maggiore leggibilità ed in prospettiva di possibili impieghi di questa soluzione, conviene definire il seguente sottoprogramma:

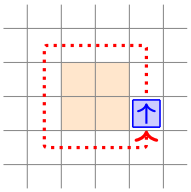
$$cavallo \stackrel{\text{def}}{=} \texttt{2FLF}$$

o quivalentemente:

$$cavallo \stackrel{\text{def}}{=} [duepassi \; \texttt{LF}]$$

□

*Problema 2.11.2* Muovere Quadretto lungo un percorso di lato pari a 4 unità, attorno ad un quadrato di lato 2 unità, come descritto nella figura che segue.



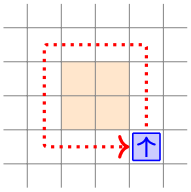
*Soluzione*. Richiamando il sottoprogramma *cavallo* definito nel problema 2.11.1, il percorso a quadrato può essere definito come segue:

$$quadrato \stackrel{\text{def}}{=} \texttt{4} \; cavallo$$

È interessante notare che nella definizione del sottoprogramma *quadrato* viene richiamato il sottoprogramma *cavallo* e, in questa fase, non interessa in che modo sia stato precedentemente definito il sottoprogramma *cavallo*. È un semplice esempio di applicazione della metodologia bottom up. Con questa

sequenza di definizioni, una chiamata al sottoprogramma *quadrato* ha l'effetto di far compiere a Quadretto un percorso su un quadrato di 4 caselle per lato, come richiesto. □

*Problema 2.11.3* Muovere Quadretto lungo un percorso di lato pari a 4 unità, attorno ad un quadrato di lato 2 unità, come descritto nella figura che segue.

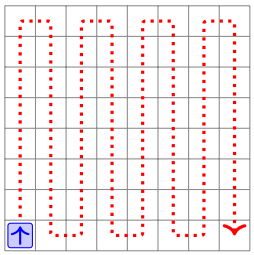


*Soluzione*. Apparentemente il problema sembra equivalente al precedente, ma la diversa posizione iniziale di Quadretto richiede una soluzione diversa in quanto il sottoprogramma *cavallo* non risulta più utile. La metodologia top-down suggerisce la seguente soluzione:

```
1: lato ≝ 3F                    ▷ percorso lungo un lato
2: quadrato ≝ 4[lato L]         ▷ percorso attorno al quadrato
```

Con questa sequenza di definizioni, una chiamata al sottoprogramma *quadrato* ha l'effetto di far compiere a Quadretto un percorso richiesto. □

*Problema 2.11.4* Muovere Quadretto lungo un percorso a serpentina, a partire dallo stato iniziale fino a raggiungere lo stato finale $(8, 1, \texttt{SOUTH})$, come si vede nella figura che segue.



*Soluzione*. Analizzando il percorso tratteggiato in figura, si può distinguere che esso è costituito da 4 componenti a forma di ⎍ connessi tra loro da 3 tratti unitari orizzontali _ che formano una curva a sinistra; ciascuna componente a forma di ⎍ è composta da 2 tratti rettilinei di 7 unità di lunghezza, connessi con un tratto unitario orizzontale _ che forma una curva a destra. Queste

considerazioni possono essere scritte in modo simbolico mediante una sequenza di definizioni che costituiscono un algoritmo che genera il percorso richiesto:

| | |
|---|---|
| 1: $t \stackrel{\text{def}}{=}$ `7F` | ▷ tratto |
| 2: $d \stackrel{\text{def}}{=}$ `RFR` | ▷ curva a destra |
| 3: $s \stackrel{\text{def}}{=}$ `LFL` | ▷ curva a sinistra |
| 4: $u \stackrel{\text{def}}{=}$ `[`$t$ $d$ $t$`]` | ▷ percorso a ⊓ |
| 5: $p \stackrel{\text{def}}{=}$ `[`$u$ `3[`$s$ $u$`]]` | ▷ percorso completo |

Con questa sequenza di definizioni, una chiamata al sottoprogramma $p$ ha l'effetto di far compiere a Quadretto il percorso a serpentina desiderato. □

*Problema 2.11.5* Usando solamente i comandi `F`, `L`, `R`, muovere Quadretto 3 *passi indietro*.

*Soluzione*. Analizziamo la frase *Fai 3 passi indietro*. Può essere scomposta in due diversi modi, come segue:

1. fai (3 passi indietro)
2. per 3 volte (fai 1 passo indietro)

Queste due diverse scomposizioni possono essere scritte, con la sintassi del linguaggio `LFR`, rispettivamente come segue:

1. `[`*inverti* `3` *avanti* *inverti*`]`
2. `3 [`*inverti* *avanti* *inverti*`]`

Queste due scomposizioni portano alle due diverse soluzioni fornite dalle seguenti due equivalenti espressioni:

$$\begin{aligned} e_1: &\quad \texttt{2R3F2L} \\ e_2: &\quad \texttt{3[2RF2R]} \end{aligned}$$

Queste due espressioni hanno le seguenti complessità:

$$\begin{aligned} compl(e_1) &= 2 + 3 + 2 = 8 \\ compl(e_2) &= 3(2 + 1 + 2) = 15 \end{aligned}$$

Da ciò si conclude che $e_1$ è più efficiente rispetto a $e_2$. L'espressione $e_2$ può essere semplificata ed ottimizzata come segue (sono state sottolineate le parti di espressione che subiscono trasformazione):

$$\begin{aligned} \texttt{3[2RF2L]} &\rightarrow \texttt{2RF}\underline{\texttt{2L2R}}\texttt{F}\underline{\texttt{2L2R}}\texttt{F2L} \\ &\rightarrow \texttt{2R}\underline{\texttt{FFF}}\texttt{2L} \\ &\rightarrow \texttt{2R3F2L} \end{aligned}$$

raggiungendo la forma di $e_1$ e confermando l'equivalenza. □

## 2.12 La ricorsione

Un caso particolare di chiamata fra sottoprogrammi si realizza quando un sottoprogramma richiama se stesso. È una situazione così importante che si è meritata un nome tutto suo: viene detta *ricorsione* e consente la realizzazione di tecniche di *programmazione ricorsiva*. Come caso particolare, queste consentono un'alternativa forma di scrittura dei controlli ciclici.

*Esempio 2.12.1 -* Un semplicissimo esempio di sottoprogramma ricorsivo è:

$$x \stackrel{\text{def}}{=} \mathtt{F}x$$

Una chiamata al sottoprogramma $x$ produce un avanzamento alla casella successiva (F) seguito da una chiamata al sottoprogramma $x$ stesso generando delle chiamate cicliche con l'effetto di avanzare Quadretto indefinitamente. Similmente, per avanzare indefinitamente Quadretto lungo un percorso quadrato di lato 3 caselle in senso antiorario si può ricorrere al sottoprogramma

$$x \stackrel{\text{def}}{=} \mathtt{2FL}x$$

□

*Osservazione*. Limitatamente ai due casi presentati nel precedente esempio sembra che i sottoprogrammi ricorsivi generino dei processi potenzialmente infiniti e di scarsa utilità. Nel primo caso Quadretto si ferma quando arriva a contatto con il bordo della scacchiera ma tenta indefinitamente di avanzare mentre nel secondo caso Quadretto gira indefinitamente lungo il bordo del quadrato. Un utilizzo più efficace della ricorsione ha bisogno di un meccanismo basato sull'uso di condizioni di chiusura che sarà presentato più avanti.

*Esempio 2.12.2 -* Un caso più articolato di ricorsione è riportato nel seguente programma che ha l'effetto di far compiere a Quadretto un percorso a serpentina come descritto nel problema 2.11.4.

---

1: $x \stackrel{\text{def}}{=} \mathtt{FT[RFR}y\mathtt{]}x$
2: $y \stackrel{\text{def}}{=} \mathtt{FT[LFL}x\mathtt{]}y$

---

Con queste due definizioni dei sottoprogrammi $x$ ed $y$, una chiamata al sottoprogramma $x$ genera il percorso desiderato. □

I sensori, grazie al fatto che informano su situazioni non note, permettono di realizzare dei meccanismi condizionali che, usati come supporto alla ricorsione, consentono di definire dei cicli condizionati finiti; in particolare permettono di realizzare il tradizionale ciclo *while* della programmazione strutturata. In generale un ciclo della forma

```
1: while C do
2:     α
3: end while
```

può essere realizzato nel linguaggio LFR, oltre alla forma diretta $\{\mathcal{C}\,\texttt{:K}\ \alpha\}$, mediante la seguente definizione ricorsiva:

$$x \stackrel{\text{def}}{=} \mathcal{C}\,\texttt{[}\alpha x\texttt{]}$$

Una chiamata ad $x$ equivale al ciclo *while* indicato sopra.

*Esempio 2.12.3 -* Vogliamo avanzare Quadretto fino ad arrivare ad una situazione di contatto, secondo lo schema che segue:

```
1: while c'è spazio davanti do
2:     avanza
3: end while
```

Nel linguaggio LFR questo ciclo può essere realizzato mediante una ricorsione come descritto nella seguente definizione:

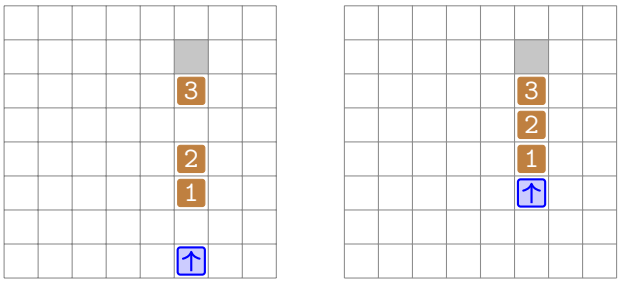
$$x \stackrel{\text{def}}{=} \texttt{S[F}x\texttt{]}$$

Una chiamata al sottoprogramma $x$ genera un ciclo che termina in quanto la $x$ nella parte destra della definizione è retta dal prefisso S che si azzera quando si verifica la condizione di contatto. □

*Esempio 2.12.4 -* Per comprimere una fila di blocchi si può ricorrere al seguente programma:

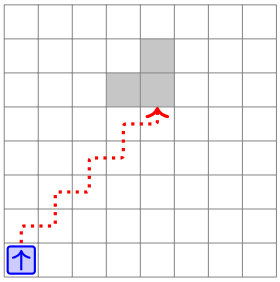
$$x \stackrel{\text{def}}{=} \texttt{P:[F}x\texttt{]}$$

La figura che segue illustra l'effetto di una chiamata al sottoprogramma $x$.




□

*Problema 2.12.1* Muovere Quadretto lungo un percorso seghettato della forma $n$[FRFL], come descritto nella figura che segue, fino ad arrivare ad una condizione di contatto con un oggetto, .



*Soluzione*. Il movimento avviene attraverso successive curve a destra ($x$) ed a sinistra ($y$). La soluzione può quindi essere costruita con delle mutue chiamate fra $x$ ed $y$, dando luogo ad un processo ciclico basato sui seguenti due sottoprogrammi:

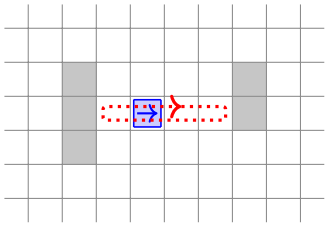
| | |
|---|---|
| 1: $x \stackrel{\text{def}}{=} \texttt{S[FR]}\,y$ | ▷ curva a destra |
| 2: $y \stackrel{\text{def}}{=} \texttt{S[FL]S}x$ | ▷ curva a sinistra |

Il procedimento ricorsivo si innesca con la chiamata $x$ e si arresta con il postfisso S$x$ presente nella definizione di $y$. I due sottoprogrammi sopra definiti possono essere inglobati nel seguente unico sottoprogramma ricorsivo:

$$x \stackrel{\text{def}}{=} \texttt{S[FR]S[FL]S}x$$

□

*Problema 2.12.2* Muovere Quadretto indefinitamente avanti-indietro fra due ostacoli posti lungo la sua direzione di movimento, come descritto nella figura che segue.

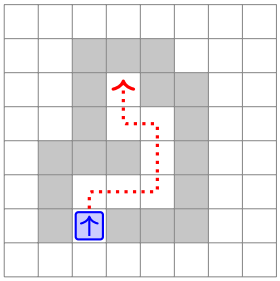


*Soluzione*. Il movimento ciclico fra i due ostacoli può essere realizzato mediante la seguente definizione ricorsiva:

$$x \stackrel{\text{def}}{=} \texttt{FT[2R]}\,x$$

In questo caso, a differenza di quanto visto nell'esempio 2.8.7, il ciclo generato dalla ricorsione non termina in quanto $x$ sulla parte destra della definizione non è retto da alcun prefisso moltiplicativo. □

*Problema 2.12.3* Percorrere un tunnel avente larghezza pari ad una casella, fermandosi al termine del tunnel, come descritto nella figura che segue.



*Soluzione*. L'idea risolutiva del procedimento si fonda sulle seguenti azioni di base (fra parentesi è riportata la corrispondente codifica in linguaggio `LFR`):

1. se c'è spazio avanza (`SF`)
2. se tocchi davanti ruota per cercare una possibile linea di avanzamento (`T[L2[TR]]`)
3. se non sei arrivato in fondo al tunnel ripeti le precedenti azioni (`S`$x$)

Pur non costituendo un algoritmo strutturato, le precedenti indicazioni si prestano ad essere facilmente codificate nel linguaggio `LFR` mediante il seguente programma ricorsivo:

$$x \stackrel{\text{def}}{=} \texttt{SFT[L2[TR]]S}x$$

□

## 2.13 Soluzione di un problema

La soluzione di molti problemi di movimento può essere ottenuta scomponendo l'intero percorso in passi elementari ciascuno dei quali, in una garantita precondizione, ha l'obiettivo di avanzare un poco, riconquistando la precondizione che servirà per il passo successivo. Per definire come svolgere il passo è in genere sufficiente esaminare un ristretto numero di situazioni che si possono presentare di volta in volta. È quanto descritto nella soluzione del seguente problema. In questa soluzione vengono utilizzati molti degli strumenti di programmazione presentati in questo capitolo.

*Problema 2.13.1* Quadretto si trova a contatto su un lato con un oggetto. Far girare Quadretto attorno all'oggetto, in senso antiorario, mantenendosi affiancato sul fianco sinistro, fino a ritornare al punto di partenza, come evidenziato nella figura 2.4.

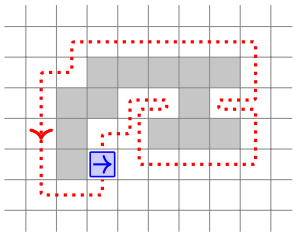


Figura 2.4: Percorso di circuitazione di un oggetto.

*Soluzione*. L'algoritmo complessivo che risolve il problema può essere formulato come segue:

**Algoritmo 6** - circuitazione di un oggetto

1: guadagna contatto sul fianco sinistro $[P_1]$
2: **while** non ha fatto un giro completo $[P_2]$ **do**
3: fai un passo elementare $[P_3]$
4: **end while**

In questo algoritmo emergono i tre sottoproblemi $P_1$, $P_2$, $P_3$ che esamineremo e risolveremo a seguire.

Nell'ipotesi che Quadretto sia a contatto con un suo lato con l'oggetto da circuitare, il sottoproblema $P_1$ può essere risolto dal seguente algoritmo:

**Algoritmo 7** - contatto sul fianco sinistro - $f$

1: **repeat**
2: ruota a sinistra
3: **until** tocchi in testa
4: ruota a destra

che, in linguaggio LFR, può essere codificato mediante il seguente sottoprogramma:

$$f \stackrel{\text{def}}{=} \texttt{\{LTK\}R}$$

Per percorrere un giro completo (sottoproblema $P_2$) si può adottare la strategia di marcare la casella di partenza e ripetere il passo elementare fino a ritornare sulla casella marcata.

Il percorso di circuitazione, come descritto nella figura 2.4, può essere realizzato mediante una sequenza di passi elementari (sottoproblema $P_3$) aventi ciascuno lo scopo di avanzare alla casella successiva, mantenendo verificata, per ciascun passo elementare, la postcondizione di contatto sul fianco sinistro. Le possibili situazioni che si possono verificare sono riportate nella figura 2.5 dove, in ciascuna diversa situazione, è descritto l'effetto del passo elementare.

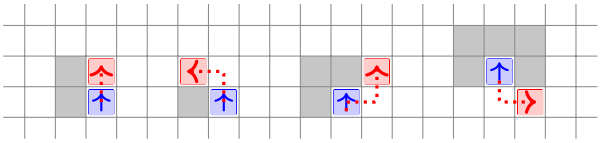


Figura 2.5: Passo elementare di avanzamento alla casella successiva, mantenendosi a contatto con il fianco sinistro: le 4 possibili situazioni.

Nelle diverse situazioni descritte nella figura 2.5 ogni passo elementare di avanzamento può essere scomposto in 3 sotto movimenti: il primo consiste in una rotazione a destra (di 0 o 1 o 2 volte) al fine di trovare un possibile verso di avanzamento, il secondo nel movimento di avanzamento alla casella successiva ed il terzo in una eventuale rotazione a sinistra per garantire la condizione di contatto sul fianco sinistro. Tutto il procedimento è descritto nell'algoritmo 8 dove, per ciascun sottoproblema, è riportata a lato la codifica in linguaggio LFR.

**Algoritmo 8** - passo elementare di circuitazione di un oggetto - $p$

| | |
|---|---|
| 1: trova verso di avanzamento | ▷ {SKR} |
| 2: avanza alla casella successiva | ▷ F |
| 3: riconquista il contatto con l'oggetto | ▷ LSF |

Ricomponendo le soluzioni dei sottoproblemi descritti nel precedente algoritmo 8, si ottiene una possibile definizione del passo elementare come segue:

$$p \stackrel{\text{def}}{=} \texttt{\{SKR\}FLSF}$$

La circuitazione dell'oggetto può essere ottenuta componendo le soluzioni dei vari sottoproblemi, secondo il tipico approccio top-down, ottenendo l'espressione complessiva

$$f\texttt{W\{}p\texttt{QK\}U}$$

□

## 2.14 Algoritmi numerici

Al paragrafo 2.3 abbiamo realizzato facilmente, sfruttando le caratteristiche del linguaggio `LFR`, le operazioni aritmetiche di addizione, sottrazione e moltiplicazione sui numeri naturali. L'operazione di divisione intera $m \div n$ fra due numeri naturali è più complessa e richiede la predisposizione di una scacchiera bidimensionale, quadrata, di $m+1$ caselle di lato, configurata con i numeri come descritto nella figura 2.6 e con Quadretto posizionato inizialmente sulla casella $(1, m+1)$ ed orientato verso sud.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| ⇧ | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Figura 2.6: La scacchiera configurata con valori per eseguire una divisione intera $m \div n$, qui limitatamente ai casi $m, n < 8$.

L'indice di riga della casella occupata da Quadretto denota il valore del dividendo che viene decrementato da $m$ a 0, mentre l'indice di colonna (riportato numericamente sulla scacchiera) denota il quoziente che man mano si incrementa di 1 quando Quadretto ha percorso verso sud un numero di caselle pari al valore $n$ del divisore. Questa strategia di calcolo è sintetizzata nell'algoritmo 9.

**Algoritmo 9** - *divisione intera* $m \div n$

**Input:** dividendo $m$, divisore $n$
**Output:** quoziente del risultato $m \div n$

```
1: assumi lo stato (1, m + 1, SOUTH)
2: loop
3:     avanza di n caselle
4:     if bloccato then
5:         exit
6:     end if
7:     passa alla colonna a destra
8: end loop
9: return  valore segnato sulla casella corrente
```

Il programma in linguaggio `LFR` corrispondente a questo schema di elaborazione è il seguente:

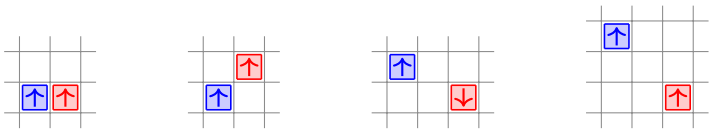
$m$`F2L{`$n$`[TKF]LFR}`

# ESERCIZI

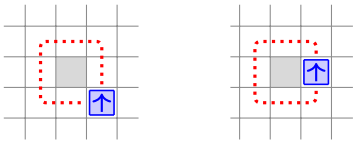
2.1 Usando solo le operazioni `F` e `L`, far eseguire a Quadretto le seguenti manovre:

- ruotare a destra
- invertire il verso di marcia
- indietreggiare di 1 passo (mantenendo il verso di marcia)
- indietreggiare di 2 passi (mantenendo il verso di marcia)

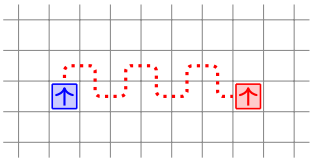
2.2 Traslare Quadretto verso destra, come descritto nelle quattro configurazioni nella figura che segue.



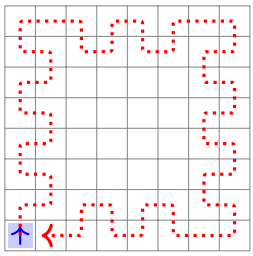
2.3 Muovere Quadretto lungo un percorso circolare quadrato attorno ad una casella, ritornando al punto di partenza, come descritto nelle due situazioni riportate nella figura che segue.



2.4 Muovere Quadretto su una traiettoria ad onda quadra, come descritto nella figura che segue.



2.5 Muovere Quadretto, a partire dallo stato iniziale $(1, 1, \texttt{NORTH})$, su una traiettoria ad onda quadra, lungo tutto il bordo fino a tornare al punto di partenza, come descritto dalla figura che segue.



2.6 Quadretto si trova nello stato iniziale $(1, 1, \texttt{NORTH})$. Stabilire lo stato finale raggiunto al termine dell'esecuzione delle seguenti espressioni:

1. `S[FTR]`
2. `3[FR2FL]`
3. `3[2[FRFL]`
4. `[12[3FR]FL]`

2.7 Scrivere delle espressioni per eseguire le seguenti transizioni di stato:

1. $(1, 1, \texttt{NORTH}) \rightarrow (8, 1, \texttt{NORTH})$
2. $(1, 1, \texttt{NORTH}) \rightarrow (8, 8, \texttt{NORTH})$
3. $(1, 1, \texttt{NORTH}) \rightarrow (6, 3, \texttt{SOUTH})$
4. $(1, 1, \texttt{SOUTH}) \rightarrow (7, 5, \texttt{EAST})$

2.8 Comparare l'effetto e l'efficienza delle seguenti due espressioni:

1. `2[TR]]`
2. `T[RTR]`

2.9 Semplificare le seguenti espressioni:

1. `SSF`
2. `STR`

2.10 Stabilire se le seguenti due espressioni sono equivalenti:

1. `L2[RDF]`
2. `DFRDFL`

2.11 Discutere la differenza fra le seguenti espressioni:

1. `DF`
2. `S[DF]`
3. `D[SF]`

2.12 Stabilire in quali situazioni il programma `TLSR` non produce alcun effetto.

2.13 Stabilire se il programma `SFTL` corrisponde a "se c'è spazio avanza altrimenti gira a sinistra".

2.14 Quadretto si trova a contatto frontale con un oggetto. Farlo ruotare finchè trovi una strada libera per intraprendere un movimento di avanzamento. Attenzione alla situazione in cui si trovi rinchiuso con un muro su ciascuno dei suoi 4 lati.

2.15 Portare Quadretto all'angolo della scacchiera situato davanti alla propria destra.

**2.16** Portare Quadretto all'angolo della scacchiera situato davanti alla propria destra.

**2.17** Stabilire il comportamento di Quadretto quando esegue l'algoritmo 8 partendo da una situazione iniziale di contatto frontale.

**2.18** Avanzare Quadretto fino ad arrivare ad una distanza di una casella dal muro.

**2.19** Spostare Quadretto ad una distanza di una mossa di cavallo degli scacchi. Quadretto può anche abbandonare alcuni tentativi nel caso si trovi ostacolato da un muro.

**2.20** Avanzare Quadretto in modo da arrivare ad una situazione di blocco (eventualmente spingendo dei blocchi) e poi ritornare alla posizione iniziale.

**2.21** Descrivere l'effetto del seguente programma:

---

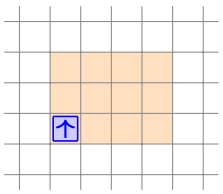
1: $a \stackrel{\text{def}}{=} \texttt{2FRF}$
2: $b \stackrel{\text{def}}{=} \texttt{2L}$
3: $p \stackrel{\text{def}}{=} \texttt{2[}a\ \ b\ \ a\texttt{]}$

---

**2.22** Sia *cavallo* $\stackrel{\text{def}}{=}$ `2FRF`. Stabilire l'effetto di `2[`*cavallo* `R]`. Usando il sottoprogramma *cavallo* eseguire la transizione di stato $(1, 1, \texttt{NORTH}) \to (8, 8, \texttt{NORTH})$.

**2.23** Stabilire se combinando i seguenti due movimenti: $a \stackrel{\text{def}}{=} \texttt{2FRF}, b \stackrel{\text{def}}{=} \texttt{L}$, si riesce ad eseguire la transizione di stato $(1, 1, \texttt{NORTH}) \to (8, 8, \texttt{NORTH})$.

**2.24** Individuare una strategia e descrivere l'algoritmo per muovere *Quadretto* su tutte le caselle di una zona rettangolare di piano di dimensioni $m \times n$, partendo da un angolo, come illustrato nella figura che segue.



**2.25** Quadretto si trova a contatto frontale con un oggetto. L'obiettivo consiste nel farlo ruotare finchè trova una strada libera per intraprendere un movimento di avanzamento. Discutere se è corretta una chiamata al sottoprogramma $x$ definito come segue:

$$x \stackrel{\text{def}}{=} \texttt{T[R}x\texttt{]}$$

**2.26** Portare Quadretto sul bordo e farlo avanzare indefinitamente lungo il bordo della scacchiera, in senso antiorario.

**2.27** Avanzare Quadretto fino a che si verifica una condizione di stallo; dopodiché ruotare a sinistra e ripetere ricorsivamente.

2.28 Avanzare Quadretto fino a superare tutte le caselle marcate, portandosi su una casella non marcata, fermandosi se eventualmente incontra un muro o un blocco.

2.29 Avanzare Quadretto fino al muro, poi ruotare a destra e continuare indefinitamente costeggiando il muro lungo il fianco sinistro.

2.30 Esprimere in formato algoritmico, usando i controlli della programmazione strutturata, la seguente porzione di codice `LFR`:

```
{S[QF:K]:K}
```

2.31 Stabilire l'effetto dei seguenti programmi ciclici; in particolare stabilire se terminano o se cadono in cicli infiniti:

1. `{SF}`
2. `S{F}`
3. `S{SF}`
4. `{TKF}`
5. `T{RSK}`
6. `{{SF}L}`

2.32 Stabilire l'effetto di una chiamata ai seguenti sottoprogrammi ricorsivi:

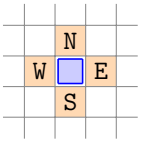
1. $x \stackrel{\text{def}}{=} \texttt{F}x$
2. $x \stackrel{\text{def}}{=} x\texttt{F}$
3. $x \stackrel{\text{def}}{=} x\texttt{SF}$
4. $x \stackrel{\text{def}}{=} \texttt{SF}x$
5. $x \stackrel{\text{def}}{=} \texttt{S}x\texttt{F}$
6. $x \stackrel{\text{def}}{=} \texttt{S[}x\texttt{F]}$

2.33 Quadretto si trova a contatto frontale con un unico blocco presente sulla scacchiera. Muovere Quadretto in modo da spingere il blocco in un angolo della scacchiera.

2.34 Un insieme di movimenti si dice *completo* se mediante una combinazione dei suoi elementi si riesce ad eseguire qualsiasi transizione di stato. Dicesi *minimale* se nessun suo sottoinsieme proprio è completo, ossia se nessun movimento dell'insieme può essere espresso come combinazione degli altri. Ad esempio, $\{\texttt{F}, \texttt{L}, \texttt{R}\}$ è completo ma non minimale in quanto $\texttt{R} \equiv 3\texttt{L}$; $\{\texttt{F}, \texttt{L}\}$ è completo e minimale. Stabilire se i seguenti insiemi di movimenti sono completi:

1. $\{\texttt{2FRF}\}$
2. $\{\texttt{F}, \texttt{FLF}\}$
3. $\{\texttt{FL}, \texttt{FR}\}$
4. $\{\texttt{FFL}, \texttt{FRF}\}$

2.35 Analizzare e discutere la gestione di Quadretto nel caso in cui non sia definita alcuna direzione di avanzamento e sia ammesso, in alternativa all'insieme di movimenti elementari $\mathcal{M} = \{\mathtt{F}, \mathtt{B}, \mathtt{L}, \mathtt{R}\}$, l'insieme di movimenti elementari $\mathcal{M}' = \{\mathtt{N}, \mathtt{E}, \mathtt{S}, \mathtt{W}\}$, dove ciascun movimento ha l'effetto di traslare Quadretto alla casella contigua posta rispettivamente a nord, est, sud, ovest rispetto alla casella occupata da Quadretto, come descritto dalla figura che segue. Discutere come, in questo caso, dovrebbero essere adattati gli algoritmi presentati in questo capitolo. Stabilire se l'insieme $\mathcal{M}'$ è *completo* (permette di raggiungere tutte le caselle del reticolo) e se è *minimale* (nessun movimento può essere espresso come combinazione degli altri).




Esprimere i comandi $\mathcal{M}'$ in funzione dei comandi $\mathcal{M}$.

# 3

# ESPLORARE

*Ossessivamente sogno di un labirinto piccolo, pulito, al cui centro c'è un'anfora che ho quasi toccato con le mani, che ho visto con i miei occhi, ma le strade erano così contorte, così confuse, che una cosa mi apparve chiara: sarei morto prima di arrivarci.*

J.L. Borges, *L'Aleph*

In questo capitolo faremo ancora riferimento a Quadretto ed alla sua programmazione per gestirne il movimento e la sua interazione con l'ambiente esterno. Nei primi paragrafi verranno riproposti alcuni contenuti ed esempi già esaminati con il linguaggio `LFR`. Ci serviremo, però, di un linguaggio più potente di quello visto nei precedenti capitoli; avremo modo di usare operazioni (aritmetiche e logiche), variabili, assegnazioni, controlli condizionali e ciclici, sottoprogrammi con argomenti; il tutto consentirà di scrivere algoritmi molto più articolati e complessi, senza più vincoli indotti dalla limitazione del formalismo descrittivo. Questi algoritmi potranno essere facilmente codificati in un linguaggio di programmazione quale, ad esempio, Python o C. È importante notare che da questo capitolo inizia a prevalere l'idea di algoritmo che viene prima (cronologicamente) e sovrasta (metodologicamente) quella di linguaggio (`LFR`, nel nostro caso), il quale viene declassato a strumento mediante il quale codificare l'algoritmo.

## 3.1 La programmazione dei movimenti

Per movimentare Quadretto utilizzeremo i seguenti comandi, espressi in notazione estesa [1] :

- *forward* : avanzamento alla casella davanti
- *backward* : indietreggiamento alla casella dietro
- *left* : rotazione a sinistra di un angolo retto
- *right* : rotazione a destra di un angolo retto

*Esempio 3.1.1* - Usando i precedenti comandi si può far fare a Quadretto un percorso a mossa di cavallo degli scacchi mediante la seguente porzione di algoritmo:

**Algoritmo 1** - percorso a mossa di cavallo - *cavallo*

```
1: for 2 times
2:     forward
3: end for
4: right
5: forward
```

Come si vede sopra, per migliorare la leggibilità e per avvicinarsi alla metodologia dei sottoprogrammi, gli algoritmi vengono spesso denominati mediante un identificatore significativo (*cavallo*) che viene poi usato per richiamare l'algoritmo innescando la sua esecuzione, costituendo di fatto la definizione di un nuovo comando. Con il comando *cavallo* di ottiene l'esecuzione delle azioni descritte nell'algoritmo stesso.
Nel linguaggio `LFR` questo algoritmo viene codificato con il programma

```
cavallo = 2FR
```

mentre in linguaggio Python sarebbe

```
def cavallo():
    for _ in range(2):
        forward()
    right()
    forward()
```

□

[1] Sfruttando la libertà di scrittura che ci concede la notazione algoritmica rispetto a quella dei linguaggi di programmazione, per allegerire la notazione scriveremo i comandi senza argomenti usando solamente l'identificatore del comando, omettendo la coppia di parentesi (); ad esempio, scriveremo semplicemente *forward* al posto di *forward*().

Una tecnica molto potente, tipica delle metodologie top-down e bottom-up, consiste nel richiamare un algoritmo all'interno di un altro.

*Esempio 3.1.2 -* Per percorrere un quadrato di lato 3 caselle si può ricorrere al seguente algoritmo che richiama l'algoritmo *cavallo*.

**Algoritmo 2** - percorso di un quadrato di 3 caselle per lato - *quadrato*

```
1: for 4 times
2:     cavallo
3: end for
```

□

Per rendere gli algoritmi adatti per risolvere delle classi di problemi, vengono usati degli argomenti (parametri).

*Esempio 3.1.3 -* Il seguente algoritmo permette di avanzare Quadretto di un numero $n$ di passi, specificato al momento della chiamata dell'algoritmo.

**Algoritmo 3** - avanza di $n$ passi - *avanza*$(n)$

**Input:** numero $n$ di passi

```
1: for n times
2:     forward
3: end for
```

L'algoritmo *avanza* può essere richiamato, ad esempio, nella forma *avanza*(5), con l'effetto di far avanzare Quadretto di 5 passi nella direzione corrente, oppure in modo generico nella forma *avanza*$(n)$ per avanzare di $n$ passi, come nel seguente algoritmo avente lo scopo di percorrere il bordo di un quadrato di lato $n$:

**Algoritmo 4** - percorso lungo un quadrato di lato $n$ - *quadrato*$(n)$

**Input:** lato $n$ del quadrato

```
1: for 4 times
2:     avanza(n)
3:     right
4: end for
```

□

*Esempio 3.1.4 -* Per ruotare in senso antiorario per un numero $n$ di rotazioni elementari si può ricorrere al seguente elementare algoritmo:

**Algoritmo 5** - rotazione a sinistra per $n$ volte - *sinistra*$(n)$

```
1: for n times
2:     left
3: end for
```

Utilizzando il precedente algoritmo *sinistra* si può realizzare il seguente algoritmo che inverte il senso di marcia:

**Algoritmo 6** - inverte il senso di marcia - *inverti*

```
1: sinistra(2)
```

□

## 3.2 Cambiamenti di stato

L'esecuzione di un programma produce un cambiamento di stato di Quadretto. Un programma $P$ può essere visto come una funzione che allo stato iniziale $S_i$ fa corrispondere lo stato finale $S_f$; in notazione funzionale:

$$S_f = P(S_i)$$

Ipotizzando che Quadretto parta dallo stato iniziale $(1, 1, \texttt{NORTH})$, ad ogni programma si può associare univocamente lo stato raggiunto eseguendo il dato programma. Si stabilisce così una corrispondenza biunivoca

$$\Psi : \mathcal{P} \to \mathcal{S}$$

fra l'insieme $\mathcal{P}$ dei programmi e l'insieme $\mathcal{S}$ degli stati raggiungibili.

L'esecuzione di un programma $P$ fa compiere a Quadretto un percorso $\lambda$ che dipende, oltre che da $P$, dallo stato iniziale $s_0$ di Quadretto. La forma del percorso $\lambda$ è individuata dal programma $P$, mentre la localizzazione (posizione ed orientamento) di $\lambda$ all'interno del reticolo dipende dallo stato iniziale $s_0$. Per indicare che un programma $P$ porta Quadretto dallo stato inziale $s_0$ allo stato finale $s_F$ lungo un percorso $\lambda$ si utilizza un *grafo di transizione di stato* come quello descritto nella figura 3.1.

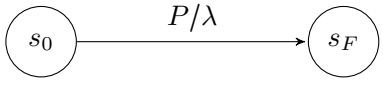



Figura 3.1: Grafo di transizione dallo stato $s_0$ allo stato $s_F$ eseguendo il programma $P$.

Un grafo di transizione di stato come quello riportato nella figura 3.1 può essere descritto in modo testuale mediante la notazione delle asserzioni di Hoare della forma

$$\{s_0\}\, P/\lambda\, \{s_F\}$$

che può essere letta come segue: a partire dallo stato $s_0$, eseguendo il programma $P$, si percorre il cammino $\lambda$ e si raggiunge lo stato $s_F$. Da questa impostazione si delineano diverse classi di problemi in cui date 3 delle 4 variabili $s_0$, $P$, $\lambda$, $s_F$ si deve determinare il valore della variabile incognita. Una importante classe di problemi corrisponde al caso in cui sono dati i valori di $s_0$, $\lambda$ e $s_F$ e si deve determinare il programma $P$ in modo da soddisfare l'asserzione $\{s_0\}\, P/\lambda\, \{s_F\}$. In molti casi il vincolo di percorrere un prefissato cammino $\lambda$ non viene imposto (e si accettano tutti i possibili percorsi). Un'altra importante classe di problemi corrisponde al caso in cui sono dati lo stato iniziale $s_0$, il programma $P$ e si deve determinare lo stato finale $s_f$ ed il percorso $\lambda$.

Un programma $P$ che assolve ad un compito è *corretto* se nelle ipotesi iniziali $A$ assunte garantisce una data post-condizione $B$; questo fatto viene indicato con il formalismo delle asserzioni di Hoare: $\{A\}P\{B\}$. In un algoritmo le condizioni di Hoare vengono solitamente espresse mediante le clausole *Require* (condizione iniziale richiesta) e *Ensure* (condizione finale raggiunta).

## 3.3 Muoversi conoscendo lo stato

Gli algoritmi, generalmente, possono essere eseguiti solo se il robot si trova all'interno di un ben definito *spazio di movimento* (ad esempio, Quadretto non può uscire dai limiti della scacchiera). In caso contrario il risultato è indefinito. Nella scrittura degli algoritmi, per evitare queste situazioni indeterminate, si preferisce richiedere esplicitamente quali sono le precondizioni necessarie alla corretta esecuzione dell'algoritmo; ciò, nell'algoritmo viene specificato mediante la clausola *Require* che specifica la condizione che deve essere verificata all'inizio dell'algoritmo. Similmente, in taluni casi, negli algoritmi viene specificata la clausola *Ensure* che esprime una postcondizione che risulterà sicuramente verificata al termine dell'esecuzione dell'algoritmo.

Per Quadretto le coordinate della posizione attuale e l'angolo attuale di avanzamento sono forniti dalle seguenti funzioni senza argomenti:

- *posx* : coordinata di riga
- *posy* : coordinata di colonna
- *angle* : angolo di avanzamento

*Esempio 3.3.1* - Per stabilire se Quadretto è sul bordo della scacchiera si può ricorrere ai sensori di stato *posx* e *posy*; la condizione è espressa dalla seguente espressione:

$$(posx = 1) \vee (posx = 8) \vee (posy = 1) \vee (posy = 8)$$

che può essere inglobata in un algoritmo:

**Algoritmo 7** - condizione di essere sul bordo - *sulbordo*

1: **return** $(posx = 1) \vee (posx = 8) \vee (posy = 1) \vee (posy = 8)$

Risulta così possibile richiamare mediante un nome la condizione, ad esempio in un'espressione della forma **if** *sulbordo* **then** ... □

*Problema 3.3.1* Orientare Quadretto secondo un dato angolo $a$.

*Soluzione*. Il problema è risolto dall'algoritmo 8.

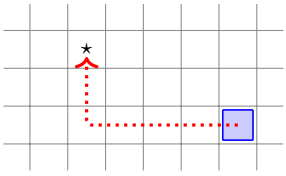
**Algoritmo 8** - orienta secondo l'angolo $a$ - $orienta(a)$

**Input:** angolo $a$ secondo cui orientarsi
**Ensure:** Quadretto è orientato secondo l'angolo $a$
1: **while** $angle \neq a$ **do**
2: *left*
3: **end while**

Notiamo che, ad alto livello come stiamo operando, non risulta necessario conoscere i valori numerici degli angoli. □

*Problema 3.3.2* Raggiungere la posizione di coordinate $(x, y)$.

*Soluzione*. La soluzione più semplice consiste nell'eseguire un movimento composto da due movimenti consecutivi lungo gli assi. Ciascuno di questi due movimenti dovrà essere preceduto da un corretto orientamento di verso. Una soluzione è fornita dall'algoritmo 9.



**Algoritmo 9** - raggiungi la posizione $(x, y)$ - $raggiungi(x, y)$

**Input:** coordinate $x$, $y$ della posizione da raggiungere
**Ensure:** Quadretto si trova sulla posizione $(x, y)$
1: $orienta(if(posx < x, \texttt{EAST}, \texttt{WEST}))$
2: $avanti(\,|posx - x|\,)$
3: $orienta(if(posy < y, \texttt{NORTH}, \texttt{SOUTH}))$
4: $avanti(\,|posy - y|\,)$

□

*Problema 3.3.3* Raggiungere l'angolo più vicino.

*Soluzione*. Osserviamo che esiste un solo angolo *più vicino* alla posizione di Quadretto. Infatti Quadretto si trova in uno dei 4 quadranti della scacchiera e ciascuno di questi quadranti ha un angolo che è più vicino. L'algoritmo si basa sull'esecuzione sequenziale dei seguenti due passi:

1. individua l'angolo più vicino $[P_1]$
2. raggiungi l'angolo individuato $[P_2]$

I due sottoproblemi $P_1$ e $P_2$ sono risolti alle linee 1-2 e 3 nell'algoritmo che segue.

**Algoritmo 10** - raggiungi l'angolo più vicino - *raggiungiAngoloVicino*

**Ensure:** Quadretto ha raggiunto l'angolo più vicino
1: $x \leftarrow$ *if*$(posx < 5, 1, 8)$
2: $y \leftarrow$ *if*$(posy < 5, 1, 8)$
3: *raggiungi*$(x, y)$

□

## 3.4 I sensori

Nelle seguenti sotto sezioni sono descritte le principali tipologie di sensori di Quadretto.

### Sensore di contatto

Una delle forme più primitive di sensore è costituita dal *sensore di contatto*. Il controllo con questo sensore avviene mediante il predicato

- *touch* : ritorna `TRUE` se Quadretto è a contatto frontalmente con un oggetto costituito dal bordo esterno della scacchiera oppure da una casella contenente un muro o un blocco.

*Esempio 3.4.1* - L'algoritmo 11 avanza Quadretto di 1 passo se non ci sono oggetti davanti.

**Algoritmo 11** - se c'è spazio avanza di 1 passo - *passo*

1: **if** ¬ *touch* **then**
2: *forward*
3: **end if**

□

### Sensore di tatto

Per riconoscere la tipologia dell'oggetto davanti si può usare il *sensore di tatto*

- *front* : ritorna uno dei valori `WALL`, `BLOCK`, `SPACE` a seconda che Quadretto sia in contatto frontale con un muro, un blocco o abbia davanti uno spazio

*Esempio 3.4.2 -* L'algoritmo 12 avanza Quadretto di un passo nel caso in cui non sia a contatto frontale con il muro.

**Algoritmo 12** - se possibile avanza - *avanza*

```
1: if front ≠ WALL then
2:     forward
3: end if
```

□

## Sensore di pressione

Per sentire se si è arrivati ad una situazione di impossibilità ad avanzare si può ricorrere ad un *sensore di pressione*, realizzato mediante il predicato

- *blocked* : ritorna `TRUE` se Quadretto è a contatto frontalmente con un muro oppure, dopo una spinta di un blocco, ha rilevato che il blocco è bloccato

*Esempio 3.4.3 -* L'algoritmo 13 avanza Quadretto, spingendo in avanti eventuali blocchi, fino ad arrivare ad una situazione di impossibilità ad avanzare.

**Algoritmo 13** - spingi i blocchi in avanti - *spingi*

```
1: while ¬ blocked do
2:     forward
3: end while
```

□

## Sensore di distanza

Quadretto è dotato di un *sensore di distanza*

- *dist* : ritorna la distanza, in unità casella, dall'oggetto o bordo davanti

Nel caso particolare in cui Quadretto si trovi a contatto frontale con l'ostacolo, il sensore di distanza fornisce il valore 0.

*Esempio 3.4.4 -* L'algoritmo 14 avanza Quadretto fino ad arrivare ad una distanza $d$ dall'oggetto davanti.

**Algoritmo 14** - ferma ad una distanza $d$ - *fermadist*$(n)$

**Input:** distanza $d$ a cui fermarsi

```
1: for dist – d times
2:     forward
3: end for
```

□

## Sensore di colore

Quadretto può avvalersi del sensore di colore

- *color* : ritorna il valore identificativo del colore della casella davanti (uno dei valori `WHITE`, `YELLOW`, `RED`, `BLACK`)

*Esempio 3.4.5 -* L'algoritmo 15 ruota Quadretto fino a trovarsi davanti ad una casella di colore $c$; nel caso in cui il colore $c$ non sia presente nelle 4 caselle contigue, Quadretto fa un giro completo e riassume lo stato iniziale.

**Algoritmo 15** - orienta verso il colore $c$ - $orientacol(c)$

**Input:** colore $c$ verso cui orientarsi
1: $k \leftarrow 1$
2: **while** $(color \neq c) \wedge (k < 5)$ **do**
3: *right*
4: $k \leftarrow k + 1$
5: **end while**

□

In molti problemi le varie tipologie di sensori vengono usati congiuntamente, come descritto nel problema che segue.

*Problema 3.4.1* Portare Quadretto nello stato iniziale $(1, 1, \texttt{NORTH})$.

*Soluzione*. Il problema proposto può essere risolto risolto combinando l'algoritmo $raggiungi(x, y)$ (Algoritmo 10) e l'algoritmo $orienta(v)$ (Algoritmo 8). Una soluzione alternativa può essere sviluppata direttamente utilizzando i sensori di stato e di contatto come descritto nell'algoritmo 16.

**Algoritmo 16** - porta nello stato $(1, 1, \texttt{NORTH})$ - *inizio*

**Ensure:** Quadretto si trova nello stato $(1, 1, \texttt{NORTH})$
1: $orienta(\texttt{SOUTH})$
2: **for** 2 **times**
3: *contatto*
4: *right*
5: **end for**

□

## 3.5 Sensori virtuali

Ogni robot è dotato di uno specifico insieme di sensori (*reali*), di diverse tipologie. Mediante opportuni algoritmi è possibile realizzare nuovi sensori (*virtuali*) basati sulle funzionalità dei sensori reali. Ad esempio, usando il sensore di contatto frontale *touch* è possibile realizzare virtualmente anche dei sensori di contatto laterale. L'algoritmo 17 realizza un sensore virtuale *toccaSinistra* che rileva se il robot è a contatto con un ostacolo sul suo fianco sinistro.

**Algoritmo 17** - *toccaSinistra*

```
1: left
2: r ← touch
3: right
4: return r
```

Similmente si possono realizzare i sensori *toccaDestra* e *toccaDietro*. I sensori *touch*, *toccaSinistra*, *toccaDestra* e *toccaDietro* possono essere inglobati in un unico sensore parametrico *tocca*($l$) dove $l \in \{\texttt{LEFT}, \texttt{FRONT}, \texttt{RIGHT}, \texttt{BACK}\}$ indica il lato di contatto da considerare (algoritmo 18). Si ha così a disposizione un nuovo *sensore virtuale tocca*($l$) che permette di stabilire se Quadretto è in contatto sul suo lato $l$.

**Algoritmo 18** - test di contatto sul lato $l$ - *tocca*($l$)

**Input:** lato $l$ sul quale testare il contatto
**Output:** `TRUE` se e solo se in contatto sul lato $l$

```
1: if l =FRONT then
2:     r ← touch
3: else if l =LEFT then
4:     left
5:     r ← touch
6:     right
7: else if l =RIGHT then
8:     right
9:     r ← touch
10:    left
11: else
12:    inverti
13:    r ← touch
14:    inverti
15: end if
16: return r
```

Aanalogamente si può realizzare un sensore virtuale *tocca*() che stabilisce se Quadretto con uno dei suoi lati è a contatto con un oggetto.

L'uso di un sensore virtuale ha il pregio di semplificare la scrittura degli algoritmi. Ma, se usato in situazioni particolari, può comportare delle inefficienze, come evidenzia l'esempio 3.5.1.

*Esempio 3.5.1 -* Per "*fare un passo a lato verso sinistra, se possibile, altrimenti mantenere la posizione ed il verso di avanzamento*" si può ricorrere al sensore virtuale *tocca*, come descritto nella seguente porzione di algoritmo:

```
1: if ¬tocca(LEFT) then
2:     left
3:     forward
4:     right
5: end if
```

Esplicitando il controllo *tocca*(LEFT) (come si vede alle linee 1-3 nella porzione di algoritmo che segue) si ottiene la seguente equivalente porzione di algoritmo:

```
1: left
2: r ← touch
3: right
4: if ¬r then
5:     left
6:     forward
7:     right
8: end if
```

Ricorrendo ai soli sensori nativi, una soluzione diretta, più efficiente, si scrive:

```
1: left
2: if ¬touch then
3:     forward
4: end if
5: right
```

□

*Problema 3.5.1* Realizzare un sensore virtuale *distanza* avente la stessa funzionalità del sensore nativo *dist* che valuta la distanza dall'ostacolo davanti.

*Soluzione*. Il sensore distanza può essere realizzato avanzando Quadretto fino ad arrivare a contatto con l'ostacolo davanti, contando quanti passi sono necessari; successivamente basta riportare alla situazione iniziale. La soluzione è descritta nell'algoritmo 19.

**Algoritmo 19** - *distanza*

**Output:** distanza dall'ostacolo davanti
1: $d \leftarrow 0$
2: **if** $\neg$ *touch* **then**
3: **while** $\neg$ *touch* **do**
4: *forward*
5: $d \leftarrow d + 1$
6: **end while**
7: *inverti*
8: *avanza*$(d)$
9: *inverti*
10: **end if**
11: **return** $d$

□

*Osservazione*. La situazione relativa alla realizzazione di un sensore virtuale è paragonabile a quella di una persona cieca che avanza e con il tatto riesce a valutare la distanza da un oggetto posto davanti oppure a quella di una persona sorda che riesce a capire le parole guardando ed interpretando il labiale della persona che sta parlando.

*Problema 3.5.2* Quadretto si trova all'interno di una stanza rettangolare avente entrambe le dimensioni maggiori di 1 ed avente un'apertura unitaria su un lato. Portare Quadretto sull'apertura in modo da chiudere la stanza (figura 3.2).

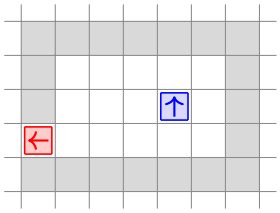


Figura 3.2: Il problema della chiusura della stanza.

*Soluzione*. La struttura di un algoritmo che risolve il problema si esprime come segue:

**Algoritmo 20** - *chiudi stanza*

**Require:** Quadretto si trova nella stanza
**Ensure:** Quadretto chiude la stanza
1: **while** $\neg$ chiusa stanza **do**
2: muoviti
3: **end while**

La condizione che la stanza sia chiusa si esprime come segue:

$$tocca(\texttt{LEFT}) \wedge tocca(\texttt{RIGHT})$$

Il cuore dell'algoritmo 20 è concentrato nell'azione *muoviti*, dettagliata nell'algoritmo 21.

**Algoritmo 21** - *muoviti*

```
1: if tocca(FRONT) then
2:    right
3: else if tocca(LEFT) then
4:    forward
5:    if ¬ tocca(LEFT) then
6:       left
7:    end if
8: else
9:    forward
10: end if
```

Innestando l'algoritmo *muoviti* nell'algoritmo *chiudi stanza* si ottiene l'algoritmo complessivo. □

## 3.6 Azioni semplici di movimento fra gli oggetti

Nella programmazione di un robot risulta spesso utile realizzare delle azioni semplici, basate sulla combinazione di alcune delle azioni elementari. Utilizzando questa tecnica, tipica delle metodologie di scomposizione in sottoproblemi, viene realizzato uno strato di software basandosi sul quale risulta più facile la realizzazione di funzionalità più avanzate. A seguire sono descritte alcune semplici azioni per muoversi fra gli ostacoli.

### Arrivare a contatto con l'oggetto davanti

Usando il sensore di contatto è possibile far avanzare Quadretto fino ad entrare in contatto con l'oggetto che si trova davanti lungo la direzione di avanzamento. Il procedimento descritto presuppone che l'oggetto si trovi lungo la direzione di avanzamento del robot, altrimenti arriva a contatto con il bordo della scacchiera; nel caso in cui Quadretto sia inizialmente già in contatto con l'oggetto, non viene svolta alcuna azione. Il procedimento è descritto nell'algoritmo 22.

**Algoritmo 22** - porta a contatto frontale con l'oggetto davanti - *contatta*

```
Ensure: tocca(FRONT)
1: while ¬ touch do
2:    forward
3: end while
```

## Inividuare l'oggetto più vicino

In varie situazioni si presenta il problema di individuare l'oggetto più vicino. Questo problema può essere risolto eseguendo delle rotazioni sul posto orientandosi verso l'oggetto individuato. La strategia consiste nell'eseguire un giro completo per determinare la distanza minima fra le 4 ed un giro parziale per mettersi con il verso corrispondente alla minima distanza determinata. A seguire è riportato un algoritmo basato su questa idea.

**Algoritmo 23** - orienta verso l'oggetto più vicino - *vicino*

**Ensure:** Quadretto è orientato verso l'oggetto più vicino
1: $d \leftarrow dist$
2: $k \leftarrow 0$
3: **for** $i$ **from** 1 **to** 3 **do**
4: *right*
5: **if** $dist < d$ **then**
6: $k \leftarrow i$
7: $d \leftarrow dist$
8: **end if**
9: **end for**
10: $destra((1 + k) \mod 4)$

## Liberarsi da un oggetto

Se Quadretto si trova a contatto frontale con un oggetto, per intraprendere un'azione di avanzamento è necessario ruotare fino a trovare una direzione di avanzamento libera. Il procedimento è descritto nell'algoritmo 24.

**Algoritmo 24** - libera il lato frontale dal contatto con l'oggetto - *libera*

**Require:** (almeno) un lato è a contatto $\wedge$ (almeno) un lato è libero
**Ensure:** $tocca(\texttt{LEFT}) \wedge \neg tocca(\texttt{FRONT})$
1: **while** *touch* **do**
2: *right*
3: **end while**

## Affiancare un ostacolo

Una tipica azione preparatoria per percorerre il perimetro di un oggetto consiste, a partire da una precondizione di contatto con l'oggetto, nell'eseguire delle rotazioni sul posto al fine di disporsi a contatto con un fianco, ad esempio il sinistro. Il problema è risolto dal semplice algoritmo 25.

**Algoritmo 25** - affianca l'oggetto sul fianco sinistro - *affianca*

**Require:** *tocca*
**Ensure:** *tocca*(LEFT)
1: **while** ¬ *touch* **do**
2: *left*
3: **end while**
4: *right*

## Strisciare lungo un oggetto

In molte situazioni si presenta l'esigenza di avanzare mantenendosi a contatto su un fianco con un oggetto. Ammettiamo la precondizione di essere a contatto sul fianco sinistro (*tocca*(LEFT)); vogliamo avanzare di 1 passo in modo che il fianco sinistro di Quadretto si sposti a contatto del successivo tratto di oggetto. Per poter eseguire in modo ciclico questa azione è necessario, dopo ogni passo, riconquistare la precondizione *tocca*(LEFT). La strategia di movimento è descrivibile come segue: *avanza di* 1 *passo e riconquista la condizione di affiancamento a sinistra.* Le situazioni da analizzare e gestire sono descritte nella figura 3.3.

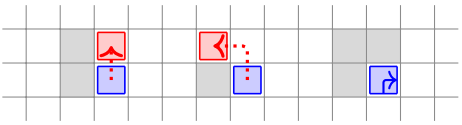


Figura 3.3: Quadretto avanza di 1 passo affiancato ad un oggetto.

Il procedimento è descritto nell'algoritmo 26.

**Algoritmo 26** - *striscia*

**Require:** *tocca*(LEFT)
**Ensure:** si è avanzati al tratto successivo ∧ *tocca*(LEFT)
1: **if** *touch* **then**
2: *right*
3: **else**
4: *forward*
5: **if** ¬ *tocca*(LEFT) **then**
6: *left*
7: *forward*
8: **end if**
9: **end if**

## 3.7 Muoversi fra gli oggetti

Combinando le azioni *contatta*, *vicino*, *libera*, *affianca* e *striscia* descritte nel precedente paragrafo si riesce a risolvere interessanti problemi di movimento fra gli oggetti, come descritto nei seguenti sottoparagrafi.

### Accostare un oggetto

In molte situazioni di movimento si rende necessaria una azione preparatoria che consiste nell'avvicinarsi all'oggetto più vicino mettendosi in contatto sul fianco sinistro con esso. L'azione è descritta nella figura 3.4.

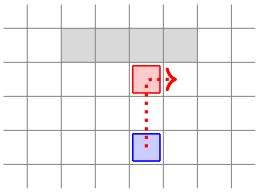


Figura 3.4: Quadretto individua l'oggetto più vicino, lo contatta frontalmente e poi si dispone a contatto sul fianco sinistro.

Questo effetto può essere ottenuto combinando le due azioni vicino e contatta come descritto nell'algoritmo 27.

**Algoritmo 27** contatta sul fianco sinistro l'oggetto più vicino - *accosta*

**Ensure:** *tocca*(`LEFT`)
1: *vicino*
2: *contatta*
3: *affianca*

### Costeggiare un ostacolo

Supponiamo che Quadretto si trovi a contatto con un oggetto e che l'oggetto sia isolato dal muro perimetrale. L'obiettivo è di costeggiare l'oggetto lungo il suo perimetro. Una tale situazione è descritta nella figura 3.5 dove la linea descrive il tragitto di costeggiamento della posizione del baricentro di Quadretto.

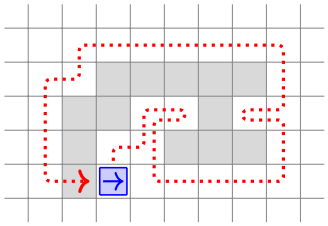


Figura 3.5: Percorso di costeggiamento di un oggetto.

*Osservazione*. Il problema è già stato affrontato, risolto e codificato in linguaggio LFR al paragrafo 2.13; qui, a differenza di quanto già fatto (dove era stata utilizzata la metodologia top-down), cerchiamo di utilizzare le azioni appena realizzate (metodologia bottom-up).

L'idea fondamentale sulla quale fondare l'algoritmo per costeggiare l'oggetto è la seguente: *affianca l'oggetto ed avanza mantenendoti a contatto con il fianco sinistro*. Lo schema generale del procedimento è descritto nell'algoritmo 28. Il ciclo infinito comporta che Quadretto costeggi l'oggetto indefinitamente.

**Algoritmo 28** costeggiare indefinitamente un oggetto - *costeggia*

**Require:** posizionamento a contatto con un oggetto
**Ensure:** costeggia l'oggetto indefinitamente
1: *affianca*
2: **loop**
3: *striscia*
4: **end loop**

## Circumnavigare un oggetto

Una variazione del problema di costeggiamento consiste nel circumnavigare l'oggetto, fermandosi, dopo un giro, alla posizione del primo contatto. A questo fine è sufficiente memorizzare la posizione di partenza e costeggiare l'oggetto fino a raggiungere la posizione iniziale. Il procedimento è descritto nell'algoritmo 29.

**Algoritmo 29** - *circumnaviga*

**Require:** posizionamento a contatto con un oggetto
**Ensure:** circumnaviga l'oggetto fino a tornare al punto di primo contatto
1: *affianca*
2: $(x, y) \leftarrow (posx, posy)$ ▷ posizione iniziale
3: **repeat**
4: *striscia*
5: **until** $(posx, posy) = (x, y)$

## Superare un oggetto

Consideriamo il problema di superare un oggetto posto davanti a Quadretto ed isolato dal bordo della scacchiera, come descritto nella figura 3.6.

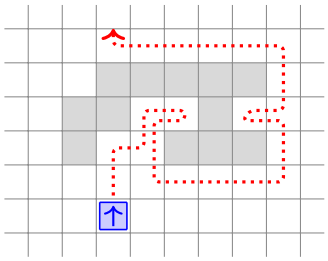


Figura 3.6: Percorso di Quadretto per superare un oggetto.

Una volta costeggiata una parte dell'oggetto, trovandosi dall'altra parte dell'oggetto, è necessario riprendere il percorso di avanzamento lungo la linea di avanzamento iniziale. Questo condizione si verifica quando Quadretto si trova con la coordinata della linea di avanzamento uguale a quella di partenza e l'altra strettamente maggiore di quella di partenza. L'obiettivo complessivo di superamento dell'oggetto può essere scomposto in sotto azioni, come descritto nell'algoritmo 30.

**Algoritmo 30** superamento di un oggetto - *supera* - ver.1

1: avanza fino ad arrivare in contatto con l'oggetto davanti (*contatta*)
2: *striscia* fino ad arrivare ad una coordinata della linea di avanzamento uguale a quella di partenza e l'altra strettamente più avanzata (rispetto alla direzione di avanzamento iniziale) di quella di partenza
3: ruota a destra

Ad un livello di dettaglio più raffinato l'algoritmo si esprime come descritto nel seguente algoritmo 31.

**Algoritmo 31** superamento di un oggetto - *supera* - ver.2

**Require:** posizionamento davanti ad un oggetto isolato dal bordo
**Ensure:** l'oggetto è stato superato e si è sulla linea di avanzamento iniziale

```
1: contatta
2: memorizza lo stato iniziale (x_0, y_0, v_0)
3: superato ← sulbordo
4: while ¬ superato do
5:    striscia
6:    if C then
7:       right
8:       superato ← TRUE
9:    end if
10: end while
```

La condizione $\mathcal{C}$ indicata alla linea 6 dell'algoritmo, indica che Quadretto si trova sulla linea di avanzamento corrispondente allo stato iniziale. Indicando con $(x_c, y_c, v_c)$ lo stato corrente di Quadretto, la condizione $\mathcal{C}$ può essere esplicitata come segue:

$$
\begin{aligned}
&((v_0 = \texttt{NORTH}) \wedge (x_c = x_0) \wedge (y_c > y_0)) \vee \\
&((v_0 = \texttt{SOUTH}) \wedge (x_c = x_0) \wedge (y_c < y_0)) \vee \\
&((v_0 = \texttt{EAST}) \wedge (x_c > x_0) \wedge (y_c = y_0)) \vee \\
&((v_0 = \texttt{WEST}) \wedge (x_c < x_0) \wedge (y_c = y_0))
\end{aligned}
$$

## 3.8 Ricercare un oggetto

Un'ampia classe di problemi rientra nella categoria di procedimenti per gli automi con stato che devono raggiungere uno stato finale obiettivo, come descritto nell'algoritmo 32.

**Algoritmo 32** - *raggiungi obiettivo*

```
1: while ¬ raggiunto obiettivo do
2:     muoviti per raggiungerlo
3: end while
```

Un esempio di applicazione di questa strategia generale è descritto nel seguente problema.

*Problema 3.8.1* Ricercare un oggetto all'interno della scacchiera, portandosi a contatto con esso.

*Soluzione*. Per risolvere questo problema si possono adottare diverse strategie, sintetizzate come segue:

1. portarsi su un angolo e percorrere un tragitto a serpentina
2. girare a spirale attorno alla posizione di partenza
3. portarsi sul bordo e girare a spirale verso il centro

Di queste diverse strategie la più semplice da realizzarsi è la prima ed è descritta nella figura 3.7.

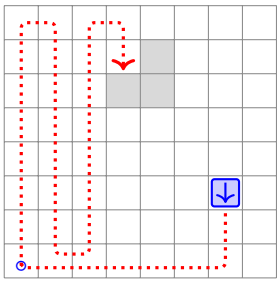


Figura 3.7: Percorso a serpentina alla ricerca di un oggetto.

La struttura generale dell'algoritmo basato su questa strategia può essere espressa come indicato nell'algoritmo 33.

**Algoritmo 33** - *ricerca oggetto*

**Require:** c'è un oggetto sulla scacchiera
**Ensure:** Quadretto è a contatto frontale con l'oggetto
1: portati nell'angolo avanti a destra $[P_1]$
2: **while** ¬ tocca ostacolo $[P_2]$ **do**
3: avanza di 1 passo su un percorso a serpentina $[P_3]$
4: **end while**

Un primo raffinamento della soluzione dei due sottoproblemi $P_1$ e $P_3$ è riportata nei due algoritmi 34 e 35, nei quali emergono i tre sottoproblemi $P_4$, $P_5$ e $P_6$.

**Algoritmo 34** - *portati nell'angolo avanti a destra*

1: **for** 2 **times**
2: avanza fino a toccare il bordo o l'oggetto $[P_4]$
3: *right*
4: **end for**

**Algoritmo 35** - *avanza di* 1 *passo su un percorso a serpentina*

1: *forward*
2: **if** tocca bordo $[P_5]$ **then**
3: fai una curva di inversione a U $[P_6]$
4: **end if**

A questo livello di dettaglio, nella soluzione dei due sottoproblemi $P_4$ e $P_6$, dove viene gestito l'avanzamento di Quadretto, è necessario prendere in considerazione l'evento che si entri in contatto con l'ostacolo ricercato. Globalmente, nella soluzione dei sottoproblemi $P_2$, $P_4$, $P_5$ e $P_6$ risulta necessario distinguere quando si entra in contatto con il bordo e quando con l'ostacolo. A questo scopo si possono utilizzare i due predicati definiti come segue (con $(x, y, v)$ è indicato lo stato attuale di Quadretto e con `NORTH`, `EAST`, `SOUTH`, `WEST` sono stati indicati i 4 versi possibili di avanzamento):

$$\begin{aligned} \mathit{toccaBordo} &\stackrel{\text{def}}{=} ((x = 1) \wedge (v = \texttt{WEST})) \vee ((x = 8) \wedge (v = \texttt{EAST})) \vee \\ &\quad ((y = 1) \wedge (v = \texttt{SOUTH})) \vee ((y = 8) \wedge (v = \texttt{NORTH})) \\ \mathit{toccaOggetto} &\stackrel{\text{def}}{=} \mathit{touch} \wedge \neg \mathit{toccaBordo} \end{aligned}$$

Il percorso a serpentina richiede di fare curve di inversione ad U alternativamente a destra ed a sinistra. A questo scopo viene utilizzata una variabile booleana *curva* che ad ogni curva inverte il suo valore. Componendo i diversi spezzoni delle soluzioni dei sotto problemi analizzati e risolti sopra, si giunge alla descrizione completa della soluzione rappresentata dall'algoritmo 36.

**Algoritmo 36** - *ricerca oggetto*

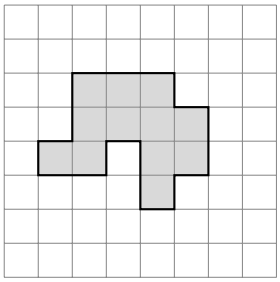
**Require:** c'è un oggetto nello spazio raggiungibile da Quadretto
**Ensure:** Quadretto è a contatto frontale con l'oggetto

```
1:  for 2 times
2:    while ¬ toccaOggetto ∧ ¬ toccaBordo do
3:      forward
4:    end while
5:    if ¬ toccaOggetto then
6:      right
7:    end if
8:  end for
9:  curva ← TRUE
10: while ¬ toccaOggetto do
11:   forward
12:   if toccaBordo then
13:     for 2 times
14:       if ¬ toccaOggetto then
15:         if curva then
16:           right
17:         else
18:           left
19:         end if
20:       end if
21:       if ¬ toccaOggetto then
22:         forward
23:       end if
24:     end for
25:     curva ← ¬ curva
26:   end if
27: end while
```

□

## 3.9 Calcolo dell'area di un oggetto

Consideriamo il problema di determinare l'area (numero di quadretti) di un oggetto presente sulla scacchiera, formato da una struttura di oggetti (blocchi o muri) contigui, nell'ipotesi che esso sia connesso e staccato dal bordo della scacchiera. Un esempio è descritto nella figura che segue.



Avendo già visto ai precedenti paragrafi come ricercare un oggetto e come circuitarlo, risulta spontaneo un approccio bottom up, basato sul seguente algoritmo:

**Algoritmo 37** - area occupata da un oggetto

1: ricerca ed affianca l'oggetto
2: circuita l'oggetto calcolando l'area da esso occupata

Questo schema di algoritmo porta inevitabilmente ad inserire i calcoli dell'area all'interno dell'algoritmo di circuitazione. Tale approccio risulta però non efficace in quanto vengono mischiati due algoritmi logicamente distinti (circuitazione e calcolo area), rendendo difficile seguirne la logica e complicando l'eventuale fase di debugging. L'approccio più pulito consiste nel rendere funzionale l'algoritmo di circuitazione in modo che ritorni come risultato il percorso effettuato, mediante una stringa del linguaggio `LFR`. Successivamente il calcolo dell'area avverrà prendendo tale stringa come dato. In questo modo il problema viene suddiviso in due parti disaccoppiate: una di natura robotica e l'altra di natura geometrica; i due algoritmi possono così essere utilizzati, in un approccio bottom up, separatamente. Questo disaccoppiamento far i due algoritmi comporta contestualmente un cambio di linguaggio: anziché parlare di "numero di quadretti occupati da un oggetto" risulta più coerente parlare di "area delimitata da un percorso".

Un algoritmo per il calcolo del percorso è il seguente:

**Algoritmo 38** - percorso di circuitazione di un oggetto - *percorso*

**Require:** contatto con l'oggetto
**Output:** stringa del percorso di circuitazione dell'oggetto

```
1: affianca
2: p ← ϵ    ▷ stringa vuota
3: while ¬ sei tornato alla posizione di partenza do
4:    if touch then
5:       right
6:       accoda R a p
7:    else
8:       forward
9:       accoda F a p
10:      if ¬ tocca(LEFT) then
11:         left
12:         forward
13:         accoda LF a p
14:      end if
15:   end if
16: end while
17: return p
```

La struttura base dell'algoritmo per il calcolo dell'area è la seguente:

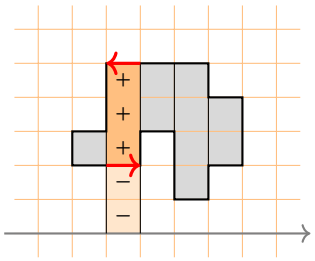
**Algoritmo 39** - area delimitata dal percorso $p$ - *area*$(p)$

**Input:** percorso $p$
**Output:** area delimitata dal percorso $p$

```
1: fissa un sistema di riferimento di posizione ed angolo: x, y, a ← 1, 1, 0
2: inizializza il valore dell'area: r ← 0
3: for c in p do
4:    aggiorna i valori di a, x, y in funzione di c    [P1]
5:    aggiorna il valore dell'area r in funzione di a, x, y    [P2]
6: end for
```

La soluzione del sottoproblema $P_1$ è elementare e viene lasciata per esercizio. La soluzione di $P_2$ è meno immediata e può essere basata sulla seguente idea: si considera l'oggetto suddiviso in fette verticali di larghezza unitaria; si fissa un asse di riferimento orizzontale; l'area di una fetta è data dal valore assoluto della differenza delle aree dei due rettangoli delimitati dall'asse di riferimento e dai due tratti unitari orizzontali sul perimetro della figura. Il valore di queste due aree viene considerato quando si percorre un tratto unitario avente direzione di avanzamento parallela all'asse di riferimento fissato. Il diverso

segno di queste due aree può essere dedotto dal fatto che in un caso ci si muove da sinistra verso destra, nell'altro in verso opposto.




Anche qui i dettagli dell'algoritmo che risolve il sottoproblema $P_2$ vengono lasciati per esercizio.

## 3.10 Uscire da un labirinto

Un *labirinto* è una struttura composta da un intricato insieme di vie; ha un ingresso, una o più uscite (o punti interni), difficili da raggiungere. I labirinti hanno una lunga tradizione storica. Uno dei più famosi è il *labirinto di Cnosso*, risalente all'epoca minoica (dal XVII al XV sec. a.C) e descritto in varie opere mitologiche. La figura 3.8 ne riporta una rappresentazione su un mosaico. Secondo la tradizione il labirinto venne fatto costruire dal re Minosse nell'isola di Creta per rinchiudervi il mostruoso Minotauro. Sempre la tradizione riporta che venne costruito da Dedalo che, a costruzione ultimata, venne fatto rinchiudere da Minosse, assieme al figlio Icaro, affinchè non potessero rivelare la piantina della costruzione.

Figura 3.8: Minotauro, Teseo e il Labirinto. - Modena, Villa di Via Cadolini, I sec. d. C.

Alcune strategie efficaci per uscire da un labirinto sono note dalla tradizione: l'eroe greco Teseo uscì dal leggendario labirinto di Cnosso usando un gomitolo di filo che aveva srotolato lungo il percorso; Pollicino riuscì ad uscire dal labiritno lasciando delle briciole di pane lungo il percorso all'andata in modo tale da poter tornare sui propri passi. Il problema di come uscire da un

labirinto è stato utilizzato anche in vari studi scientifici sul comportamento e l'apprendimento degli animali. Il problema è stato riconsiderato ultimamente in connessione ai robot, tantoché è diventato un classico problema delle gare di robotica.

Abbandonando la tradizione mitologica, un labirinto è un oggetto matematico che può essere rappresentato mediante un grafo e studiato ed analizzato mediante l'omologa teoria. Il problema di base di un labirinto consiste nel determinare delle strategie che permettano di trovare in modo efficiente la via da percorrere. Se si conosce la pianta del labirinto è possibile adottare un approccio brute-force, percorrendo in modo esaustivo tutti i possibili cammini, trovando quello porta all'uscita. Non conoscendo la pianta bisogna adottare una strategia diversa. Un metodo semplice e sicuro consiste nell'appoggiare la mano destra (o la sinistra) alla parete destra del labirinto (o rispettivamente alla parete sinistra) all'entrata del labirinto, e camminare senza staccare mai la mano dalla parete scelta, fino a raggiungere una delle eventuali altre uscite, o il punto di partenza. La regola della mano (destra o sinistra, equivalentemente) pur non garantendo il percorso più breve, permette di individuare con certezza un'uscita dal labirinto, nell'ipotesi che essa esista e che il percorso che compone il labirinto sia semplicemente connesso. La garanzia di successo di questo procedimento è fornita da alcuni risultati teorici sugli spazi topologici semplicemente connessi caratterizzati dal fatto che per ogni punto si può definire una curva chiusa passante per il punto ed è possibile deformare con continuità la curva all'interno dello spazio, fino a renderla il punto stesso. È possibile dimostrare che ogni spazio semplicemente connesso può essere deformato con continuità fino a diventare un cerchio: il labirinto è quindi topologicamente equivalente a una stanza circolare con una o più porte. È quindi evidente che seguendo la parete interna della stanza si giungerà inevitabilmente a una porta.

Alla fine del XIX secolo i matematici francesi G. Tarry e M. Trémaux idearono un algoritmo in grado di risolvere qualsiasi labirinto. L'algoritmo è descritto a seguire (algoritmo 40).

**Algoritmo 40** - metodo di Tremaux

1: Quando ti trovi in un nodo prendi il ramo più a destra. Se arrivi a un vicolo cieco, ritorna sui tuoi passi fino all'ultimo nodo e prendi il ramo più a destra tra quelli ancora inesplorati.

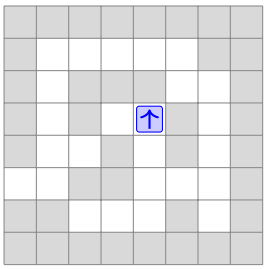
Il precedente algoritmo può essere formulato anche nella seguente equivalente versione, nota come *regola della mano destra* (algoritmo 41).

**Algoritmo 41** - metodo della mano destra

1: Avanza toccando con la mano destra il muro sul fianco destro.

Nei due algoritmi 40 e 41 si può usare il ramo più a sinistra e la mano sinistra al posto del ramo più a destra e la mano destra, ottenendo algoritmi equivalenti.

Adattiamo ora l'algoritmo 41 al caso in cui Quadretto, partendo da una posizione interna al reticolo, deve uscire dal labirinto avanzando fino a raggiungere una casella sul bordo. Un esempio di situazione è descritta nella figura che segue.



L'algoritmo si scrive:

**Algoritmo 42** - *esci dal labirinto*

**Require:** Quadretto tocca il labirinto $\wedge$ il labirinto è connesso
**Ensure:** Quadretto è a contatto frontale con l'oggetto
1: **while** $\neg$ *toccaBordo* **do**
2: *striscia*
3: **end while**

La condizione che il labirinto sia connesso è essenziale: se si togliesse, ad esempio, la casella di coordinate $(3, 3)$ Quadretto continuerebbe a girare attorno allo spezzone centrale di labirinto, senza mai trovare la strada di uscita.

## ESERCIZI

3.1 A partire dallo stato $(1, 1, \texttt{NORTH})$, far fare a Quadretto un percorso lungo tutto il bordo della scacchiera e ritornare al punto di partenza.

3.2 Muovere Quadretto dalla posizione corrente alla posizione $P = (x_1, y_1)$ passando per un fissato punto $Q = (x_2, y_2)$.

3.3 Muovere Quadretto alle posizioni $P = (x_1, y_1)$ e $Q = (x_2, y_2)$ per il percorso più breve (è indifferente l'ordine con il quale si raggiungono le due posizioni).

3.4 Eseguire la transizione dallo stato attuale ad un determinato stato $(x, y, v)$.

3.5 Scrivere delle condizioni corrispondenti alle seguenti situazioni di Quadretto:

1. si trova sul bordo
2. si trova sul bordo con la parte frontale a contatto con il bordo
3. si trova in un angolo
4. si trova in un angolo ed è rivolto verso il bordo

3.6 Muovere Quadretto indefinitamente avanti-indietro fra due oggetti (blocchi o muri) posti sulla direzione di movimento (uno da una parte ed uno dall'altra rispetto al verso di avanzamento).

3.7 Muovere Quadretto a contatto frontale con l'oggetto davanti.

3.8 Muovere Quadretto a contatto frontale con l'oggetto più vicino.

3.9 Determinare la minima distanza di Quadretto dal bordo (0 se è sul bordo).

3.10 Stabilire se sulla scacchiera sono presenti oggetti.

3.11 Assumendo l'ipotesi che la scacchiera sia sgombra da oggetti, portare Quadretto in un angolo usando:

1. il solo sensore di contatto
2. i sensori di stato
3. il sensore di distanza

3.12 Muovere Quadretto nell'angolo più vicino.

3.13 Nell'ambiente ci sono alcune caselle-muro; portare Quadretto (ovunque si trovi inizialmente) ad una data casella (evitando i muri presenti nell'ambiente).

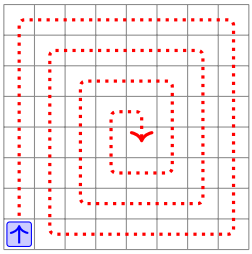
3.14 Traslare Quadretto di $n$ passi in un dato verso $v$.

3.15 Traslare Quadretto di $n$ passi a sinistra (a destra).

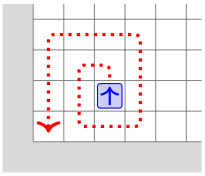
3.16 Avanzare Quadretto di $n$ passi, fermandosi prima se si entra in contatto con un oggetto.

3.17 Portare Quadretto in una nicchia del muro, in modo che abbia i tre lati davanti a contatto con il muro. Si assuma l'ipotesi che ci sia almeno una nicchia accessibile.

3.18 Muovere Quadretto lungo un percorso a spirale verso l'interno, a partire dallo stato iniziale $(1, 1, \texttt{NORTH})$.



3.19 Muovere Quadretto lungo un percorso a spirale verso l'esterno come descritto nella figura che segue, arrestando quando si arriva a contatto con il bordo.



3.20 Muovere Quadretto di un moto casuale fino a quando arriva a contatto con il bordo.

3.21 Realizzare un sensore virtuale *tocca* che ritorna `TRUE` se e solo se Quadretto è a contatto con un oggetto con uno dei suoi 4 lati.

3.22 Realizzare un sensore virtuale di *distanza* che rileva la distanza dall'oggetto posto davanti. Suggerimento: avanzare fino ad arrivare a contatto con il muro, contando i passi, e poi ritornare nella situazione di partenza.

3.23 Adattare l'algoritmo 20 (*chiudi stanza*) in modo da gestire le situazioni in cui la stanza abbia una o entrambe le dimensioni pari ad una casella.

3.24 Spiegare perché l'algoritmo 20 (*chiudi stanza*) non funziona se viene impostato secondo la seguente strategia:

```
1: avanza fino a contattare il muro
2: ruota a destra
3: striscia lungo il muro fino a chiudere la porta
```

3.25 Stabilire l'effetto dell'algoritmo 26 (*striscia*) nel caso in cui non sia vera la precondizione *tocca*(`LEFT`).

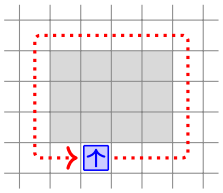
3.26 Stabilire l'effetto dell'algoritmo 28 (*costeggia*) se l'oggetto da costeggiare è a contatto con il muro perimetrale.

3.27 Stabilire l'effetto dell'algoritmo 42 (*esci dal labirinto*) nel caso in cui la scacchiera sia completamente libera da oggetti.

3.28 Analizzare l'algoritmo 42 (*esci dal labirinto*) e stabilirne l'effetto nel caso in cui la scacchiera sia completamente sgombra da oggetti.

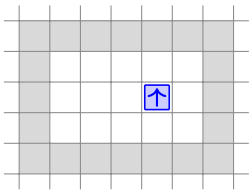
3.29 Adattare l'algoritmo 42 (*esci dal labirinto*) in modo che spinga in avanti (fino ad un angolo) eventuali blocchi incontrati lungo il percorso.

3.30 Quadretto si trova a contatto con un muro rettangolare. Far circumnavigare Quadretto attorno al muro, a contatto con il muro sul lato sinistro, fino a tornare al punto di partenza.

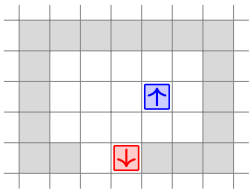


3.31 Nella situazione dell'esercizio 3.30, determinare il perimetro e l'area del muro rettangolare.

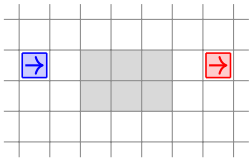
3.32 Quadretto si trova all'interno di un recinto rettangolare. Determinare il perimetro (lunghezza del perimetro interno) e l'area (numero di celle all'interno del recinto).



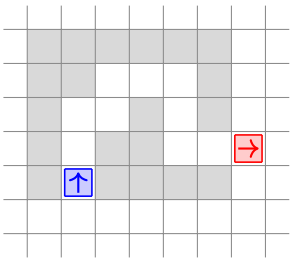
3.33 Quadretto si trova all'interno di una stanza rettangolare di dimensioni $m \times n$ unità ed avente su un lato un'apertura di 1 o più unità. Far uscire Quadretto dalla stanza, portandolo sul bordo. Analizzare e risolvere il problema nel caso in cui non si conoscano le dimensioni della stanza.



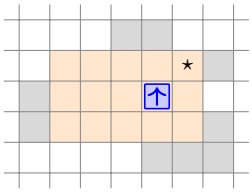
3.34 Quadretto si trova davanti ad un muro rettangolare. Portare Quadretto dall'altra parte del muro, in una posizione speculare al muro rispetto alla posizione di partenza.



3.35 Quadretto si trova all'interno di un tunnel lineare di larghezza unitaria. Farlo uscire dal tunnel.



3.36 Determinare l'area di una stanza rettangolare disarrocata, avente almento un elemento non angolare di muro su ciascun lato. In pratica si tratta di determinare il più grande rettangolo libero contenente la posizione attuale di Quadretto. Attenzione alla posizione iniziale critica denotata con $\star$ nella figura.




3.37 All'interno della scacchiera è presente un oggetto di forma generica, non appoggiato al bordo. Individuare l'oggetto, portando Quadretto a contatto con l'oggetto. Analizzare e risolvere il problema nel caso in cui la stanza sia completamente sgombra da oggetti, fermando Quadretto nel momento in cui si sia riconosciuto che nella stanza non è presente alcun oggetto. Generalizzare il problema ai casi in cui ci possano essere più oggetti all'interno della stanza ed al caso in cui gli oggetti possano essere appoggiati al bordo.

3.38 Nell'ambiente di Quadretto ci sono alcune caselle colorate con diversi colori (non ci sono caselle-muro); contare le caselle di un dato colore.

3.39 All'interno della scacchiera, staccato dal muro perimetrale, c'è un oggetto connesso e circumnavigabile. Calcolare la lunghezza del perimetro esterno dell'oggetto.

3.40 All'interno della scacchiera ci sono degli oggetti che occupano una singola casella e sono staccati dal muro perimetrale e fra di loro (non si toccano neanche sugli spigoli); determinare il numero di oggetti presenti.

3.41 Raggiungere una data posizione, evitando eventuali oggetti presenti sulla scacchiera.

3.42 Visitare e contare tutte le caselle libere accessibili da Quadretto.

3.43 Sulla scacchiera è presente un solo blocco, staccato dal bordo. Spingerlo:

1. contro il bordo
2. in un angolo
3. nell'angolo più vicino

3.44 Sulla scacchiera sono presenti due blocchi, non entrambi in un angolo. Spingerli in modo da farli toccare fra loro su un lato.

3.45 Sulla scacchiera sono presenti quattro blocchi staccati dal bordo. Raggrupparli in modo da formare un agglomerato connesso staccato dal bordo.

3.46 Sulla scacchiera sono presenti quattro blocchi separati fra loro. Spingerli in modo che ciascuno di essi occupi un angolo della scacchiera.

3.47 Sulla scacchiera sono presenti dei blocchi staccati dal bordo e separati fra loro. Non sono presenti muri. Spingere i blocchi in modo da raggrupparli in modo da formare un quadrato (di lato 2 caselle).

3.48 Contare i blocchi presenti sulla scacchiera. Si assuma l'ipotesi che tutti i blocchi siano staccati dal bordo e che non siano presenti muri.

# 4

# TRACCIARE

*La geometria della Tartaruga è uno stile di geometria diverso dagli altri, come lo stile assiomatico d'Euclide e lo stile analitico di Cartesio erano anch'essi differenti l'uno dall'altro. Lo stile d'Euclide è logico, quello di Cartesio è algebrico. Lo stile della geometria della tartaruga è informatico.*

S. Papert, *Mindstorm*

Il protagonista di questo capitolo è Tartaruga, un automa che costituisce un'evoluzione dell'automa Puntino, già visto al cap. *Muovere*. Rispetto a Puntino, Tartaruga è in grado di ruotare di un angolo generico (e non solo di un angolo retto) ed è in grado di avanzare di un tratto di una generica lunghezza. Questa evoluzione contribuisce ad evidenziare gli aspetti geometrici delle figure che vengono prodotte dal movimento. I contenuti corrispondono alla cosiddetta *geometria della tartaruga*, caratterizzata da un automa che si muove su un foglio, lasciando una *traccia* che genera un disegno.

All'interno di molti linguaggi di programmazione è predisposta una componente di linguaggio imperativo costituita dai comandi della grafica della tartaruga, similmente a quanto si ritrova anche nel linguaggio Logo. Per descrivere gli algoritmi viene utilizzata la notazione introdotta al cap. *Esplorare*; questa notazione trova la sua più aderente codifica con il linguaggio Python.

L'ambiente della tartaruga, nonostante l'apparenza, è molto ricco e versatile: la tartaruga diventa un pretesto su cui realizzare contesti per il disegno, per la geometria e per il calcolo.

## 4.1 Una tartaruga che disegna

Consideriamo un automa, noto con il nome di *tartaruga*, che, muovendosi, lascia una traccia disegnando delle figure. Il piano di movimento della tartaruga è costituito da un foglio virtuale rappresentato dal video del calcolatore; la tartaruga è posizionata sul foglio ed ha una propria direzione di avanzamento (modificabile mediante appositi comandi). Alla tartaruga è attaccata una penna che, muovendosi assieme alla tartaruga, lascia un segno mediante il quale viene generato il disegno. La tartaruga esegue gli ordini che le vengono impartiti: può essere fatta ruotare sul posto, a sinistra ed a destra, e fatta avanzare. A seguire sono schematicamente riportati alcuni comandi di base della grafica della tartaruga che saranno utilizzati più avanti:

| | |
|---|---|
| $forward(n)$ | avanza di $n$ passi |
| $left(a)$ | gira a sinistra di un angolo $a$ |
| $right(a)$ | gira a destra di un angolo $a$ |

Gli angoli di rotazione sono espressi in gradi sessagesimali. Questi semplici comandi consentono il disegno di figure geometriche, come descritto nell'esempio 4.1.1.

*Esempio 4.1.1 -* Il disegno di un quadrato di lato 50 unità può essere generato mediante il seguente schema di ripetizione di una sequenza di comandi:

```
1: for 4 times
2:     forward(50)
3:     left(90)
4: end for
```

Sono predisposti anche i seguenti comandi per gestire l'angolo di avanzamento della tartaruga (riferito ad un prefissato sistema di orientamento assoluto):

| | |
|---|---|
| $getang()$ | angolo di avanzamento attuale |
| $setang(\alpha)$ | imposta ad $\alpha$ l'angolo di avanzamento |

Se si vuole tracciare una linea che inizia in un punto diverso dalla posizione corrente della tartaruga si possono adottare due strategie alternative:

1. sollevare ed abbassare la penna sul foglio; naturalmente, un movimento con la penna abbassata ha l'effetto di lasciare una linea mentre un movimento con la penna alzata non produce alcun effetto grafico. Questa possibilità si fonda sull'uso dei seguenti due comandi:

   | | |
   |---|---|
   | $penup()$ | alza la penna |
   | $pendown()$ | abbassa la penna |

2. far saltare la tartaruga in avanti di un dato numero di passi; il comando corrispondente è il seguente:

   | | |
   |---|---|
   | $jump(n)$ | salta in avanti di $n$ passi |

*Osservazione*. Per questioni di efficienza e di pulizia grafica, negli esempi, nei problemi e negli esercizi che seguiranno, ammetteremo tacitamente il vincolo che la tartaruga non disegni sopra un tracciato disegnato in precedenza.

*Osservazione*. L'ambiente della tartaruga è primitivo rispetto a qualsiasi forma di concettualizzazione geometrica in quanto non richiede alcuna specifica conoscenza delle proprietà dello spazio geometrico ma solamente una consapevolezza del rapporto del proprio corpo rispetto allo spazio esterno. La caratteristica interessante e peculiare della geometria della tartaruga consiste nella possibilità di poter evitare il ricorso ad un sistema di riferimento di coordinate, in quanto ogni azione grafica fa riferimento alla posizione e direzione attuale della tartaruga. A differenza della tradizionale impostazione assiomatica della geometria euclidea, viene adottata un'impostazione costruttiva; ad esempio, il quadrato non viene definito dichiarativamente mediante una proprietà caratterizzante (come avviene nella geometria euclidea o cartesiana) ma viene *costruito*: la circonferenza viene specificata mediante un algoritmo che la disegna.

## 4.2 Istruire la tartaruga

L'ambiente grafico predisposto mediante i comandi grafici descritti nel paragrafo precedente può essere arricchito mediante la definizione di nuovi comandi da parte dell'utente. Questa possibilità può essere vista come un meccanismo per istruire la tartaruga, insegnandole nuovi comandi. Questa idea viene concretizzata mediante un algoritmo che viene poi codificato in linee di programma mediante un sottoprogramma.

*Esempio 4.2.1* - Per disegnare un generico esagono poligono regolare di $n$ lati di lunghezza $l$ si può definire il seguente algoritmo.

**Algoritmo 1** - $poligono(n, l)$ : poligono regolare di $n$ lati di lunghezza $l$

**Input:** numero $n$ di lati, lunghezza $l$ del lato

1: **for** $n$ **times**
2: $forward(l)$
3: $left(360/n)$
4: **end for**

Per disegnare un esagono di lato 50 basta richiamare il precedente algoritmo mediante l'istruzione

$$poligono(6, 50)$$

□

## 4.3 Figure ripetitive

Per sviluppare un algoritmo per disegnare una figura composta da un elemento grafico che si ripete è conveniente svolgere delle fasi di analisi che portano ad individuare i seguenti elementi:

1. parametri che definiscono la classe delle figure
2. elementi che si ripetono
3. struttura dello schema ripetitivo

*Esempio 4.3.1* - Le stelle costituiscono delle tipiche strutture ripetitive che si prestano ad essere generate mediante la grafica della tartaruga. A seguire è riportato l'algoritmo per disegnare una stella a 5 punte con lati di lunghezza $l$, posti lungo i prolungamenti di un pentagono regolare.

In questo caso:

1. la classe delle figure è individuata dalla lunghezza $l$ del lato della stella
2. gli elementi grafici che si ripetono sono le "punte" della stella
3. la stella è composta da una sequenza di punte, ciascuna ruotata rispetto alla precedente

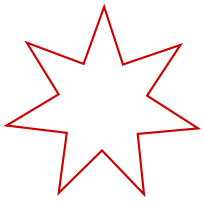
In base a queste considerazioni si arriva al seguente algoritmo risolutivo. I valori 144 e 72 degli angoli di rotazione sono ricavabili in base a semplici considerazioni sugli angoli.

**Algoritmo 2** - $stella(l)$ : stella a 5 punte di lato $l$

**Input:** lunghezza $l$ del lato

```
1: for 5 times
2:     forward(l)
3:     right(144)
4:     forward(l)
5:     left(72)
6: end for
```

*Problema 4.3.1* Generalizzare il precedente esempio, disegnando una generica stella, come quella descritta nella figura che segue.



*Soluzione*. Una generica stella è individuata dal numero $n$ di punte, dalla lunghezza $l$ del lato e dall'angolo $a$ in punta. L'elemento che si ripete è costituito dalla *punta*. (la figura precedente illustra una stella a 7 punte). Il procedimento per disegnare una stella ad $n$ punte si fonda sulle seguenti considerazioni: indicando con $\alpha = 180 - a$ l'angolo di rotazione a sinistra che la tartaruga esegue trovandosi in una punta della stella e con $\beta$ l'angolo di rotazione a destra quando la tartaruga si trova su uno spigolo interno, deve valere la relazione $n\alpha - n\beta = 360$ da cui si ricava: $\beta = \alpha - 360/n$. Il procedimento è descritto nell'algoritmo 3.

**Algoritmo 3** - *stella*$(n, l, a)$ : stella a $n$ punte di lato $l$ ed angolo $a$ in punta

**Input:** numero $n$ di punte, lunghezza $l$ del lato, angolo $a$ in punta

1: $\alpha \leftarrow 180 - a$
2: $\beta \leftarrow \alpha - 360/n$
3: **for** $n$ **times**
4: *forward*$(l)$
5: *left*$(\alpha)$
6: *forward*$(l)$
7: *right*$(\beta)$
8: **end for**

□

## 4.4 Figure ricorsive

La ricorsione trova un fertile terreno di applicazione nell'ambito della grafica. Gli esempi che seguono illustrano alcune famose figure ricorsive.

*Esempio 4.4.1* - Un altro famoso esempio di figura ricorsiva è costituito dalla *curva di Von Kock*, definita come segue: la curva di base $l$ ed ordine 0 è costituita da un segmento di lunghezza $l$; una curva di ordine 1 viene costruita dividendo il segmento di base in 3 parti, eliminando la parte centrale, sulla quale viene innalzato un triangolo equilatero; in generale una curva di ordine superiore viene costruita replicando questo procedimento, innalzando un triangolo equilatero sulla parte centrale di ogni tratto rettilineo della curva di ordine precedente. La curva di Von Kock ha la caratteristica, di particolare interesse nell'ambito dell'Analisi Matematica, di generare, al tendere all'infinito dell'ordine $n$ di ricorsione, una funzione continua che non è derivabile in alcun punto.

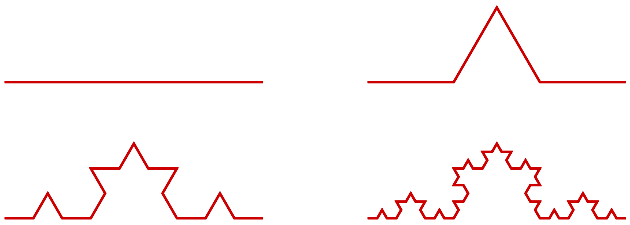


Figura 4.1: Istanze della curva di von Kock di ordine 1, 2, 3 e 4.

L'algoritmo 4 disegna una curva di Von Kock, mediante i comandi della grafica della tartaruga.

**Algoritmo 4** - $vonKock(l, n)$ : curva di lunghezza $l$ ed ordine $n$

**Input:** $l$, $n$

```
1:  if n = 0 then
2:      forward(l)
3:  else
4:      vonKock(l/3, n − 1)
5:      left(60)
6:      vonKock(l/3, n − 1)
7:      right(120)
8:      vonKock(l/3, n − 1)
9:      left(60)
10:     vonKock(l/3, n − 1)
11: end if
```

□

*Esempio 4.4.2* - Gli *alberi* sono delle particolari figure che si prestano naturalmente ad essere definite in modo ricorsivo. La forma più semplice di albero è costituita dagli *alberi binari* in cui ogni ramo genera due altri rami, di lunghezza pari alla metà del ramo padre che li ha generati e divaricati ad angolo retto. Indicando con $T(l, n)$ un generico albero binario di lunghezza del fusto $l$ ed ordine di ramificazione $n$, un albero binario può essere descritto ricorsivamente dalla figura 4.2.

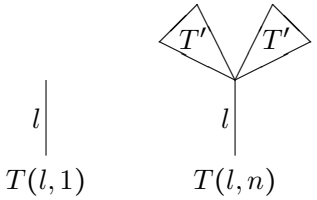



Figura 4.2: Schema della definizione ricorsiva di un *albero binario*; $T'$ è un sottoalbero definito come segue: $T' = T(l/2, n - 1)$.

Nella figura 4.3 è riportato un albero binario di lunghezza 2 centimetri (altezza del fusto dell'albero) e di ordine 4.

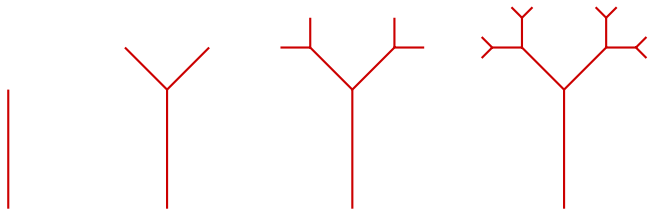


Figura 4.3: Le prime istanze della successione di alberi binari di ordine $1, 2, 3, 4$.

L'algoritmo 5 disegna, usando la grafica della tartaruga, un albero mediante una procedura ricorsiva derivata direttamente dalla precedente definizione. Le istruzioni $8 - 12$ servono per riportare la tartaruga alla radice dell'albero in modo da raccordare correttamente i due sotto alberi che, ad ogni livello, si biforcano a destra ed a sinistra del fusto.

**Algoritmo 5** - $albero(l, n)$ : albero di lunghezza $l$ ed ordine $n$

**Input:** lunghezza $l$, ordine $n$

```
1:  forward(l)
2:  if n > 0 then
3:      left(45)
4:      albero(l/2, n − 1)
5:      right(90)
6:      albero(l/2, n − 1)
7:  end if
8:  penup()
9:  left(180)
10: forward(l)
11: right(180)
12: pendown()
```

□

*Esempio 4.4.3* - Nell'ambito dell'Analisi Matematica è famosa la seguente *curva di Hilbert-Peano*, nota con il nome dei matematici che l'hanno scoperta ed analizzata. L'importante caratteristica che la contraddistingue consiste nel generare, al tendere all'infinito dell'ordine di ricorsione, una curva che *riempie* una porzione di piano. Alcune istanze di questa curva sono riportate nella figura 4.4. La curva di Hilbert-Peano può essere disegnata mediante un procedimento ricorsivo descritto nell'algoritmo 6; il parametro $k = 1$ genera una curva a destra rispetto alla direzione attuale, il valore $k = {}^{-}1$ una curva a sinistra.

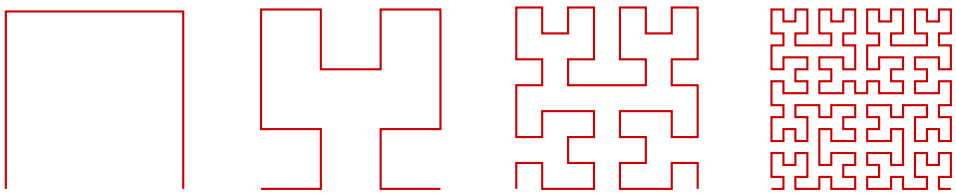


Figura 4.4: Istanze della curva di Hilbert-Peano di ordine 1, 2, 3 e 4 (per $k = 1$).

**Algoritmo 6** - $hilbert(l, n, k)$ : curva di di lunghezza $l$, ordine $n$ e verso $k$

**Input:** lunghezza $l$, ordine $n$, verso $k$

1: **if** $n > 0$ **then**
2: $left(k * 90)$
3: $hilbert(l, n-1, -k)$
4: $forward(l)$
5: $right(k * 90)$
6: $hilbert(l, n-1, k)$
7: $forward(l)$
8: $hilbert(l, n-1, k)$
9: $right(k * 90)$
10: $forward(l)$
11: $hilbert(l, n-1, -k)$
12: $left(k * 90)$
13: **end if**

*Esempio 4.4.4 -* La *curva del drago* fu scoperta dal fisico della NASA John Heighway. Fu successivamente analizzata dai matematici e teorici dei computer Donald Knuth e Chandler Davis, con particolare riferimento alle rappresentazioni dei numeri. Il modo più semplice per descriverla si basa su un procedimento grafico ricorsivo, simile al metodo usato per la generazione della curva di Von Kock: si inizia con un segmento che costituisce la curva di ordine 0; ad ogni passo successivo della ricorsione si sostituisce ciascun segmento della curva con due segmenti adiacenti che formano un angolo retto, alternativamente dalle due bande della curva (vedi figura 4.5).

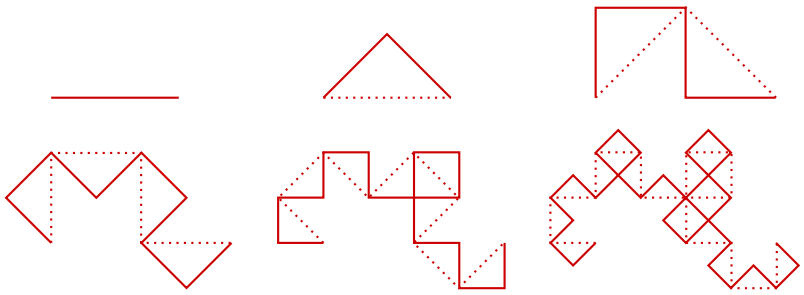


Figura 4.5: Passi della generazione di alcune istanze della curva del drago, dall'istanza di ordine 0 all'istanza di ordine 5; la linea tratteggiata denota la curva di supporto ottenuta al passo precedente.

La figura 4.6 descrive la curva del drago di ordine 6 senza le linee di supporto dell'istanza di ordine 5 e con gli angoli arrotondati per evitare l'apparente incrocio delle linee tangenti fra loro in alcuni spigoli.

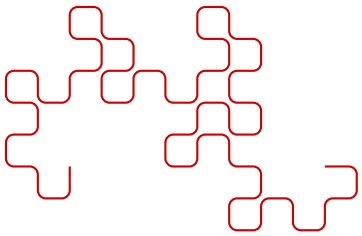


Figura 4.6: Curva del drago di ordine 6.

Il procedimento sopra illustrato suggerisce direttamente l'algoritmo 7.

**Algoritmo 7** - $drago(l, n, k)$ : curva di lunghezza $l$, ordine $n$ e verso $k$

**Input:** lunghezza $l$, ordine $n$, verso $k$: 0: a sinistra, 1: a destra

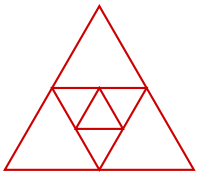
1: **if** $n = 0$ **then**
2: $forward(l)$
3: **else**
4: $left(45 + k * 270)$
5: $drago(l, n - 1, 0)$
6: $right(90 + k * 180)$
7: $drago(l, n - 1, 1)$
8: $left(45 + k * 270)$
9: **end if**

Un modo pratico per generare la curva del drago consiste nel prendere un foglio di carta ed eseguire dei successivi piegamenti del foglio a metà: il profilo del foglio, opportunamente dispiegato in modo da formare degli angoli retti in corrispondenza di ciascun piegamento, assumerà, di profilo, la forma di una curva del drago.

## 4.5 Ripetere e ricorrere

*Ripetere* e *ricorrere* sono due meccanismi trasversali che risultano potenzialmente equivalenti, nel senso che quello che può essere descritto con l'uno è possibile farlo con l'altro. Nonostante l'equivalenza fra i due meccanismi, dal punto di vista pratico sono sostanzialmente differenti, sia dal punto di vista all'approccio alla soluzione dei problemi sia dal punto di vista della complessità ed efficienza del processo generato.

*Esempio 4.5.1 -* Vogliamo disegnare con la grafica della tartaruga una sequenza di triangoli equilateri annidati, ciascuno avente i vertici sul punto medio del triangolo in cui è inscritto.



La figura può essere scomposta e considerata formata da una *sequenza di triangoli* ciascuno dei quali avente per vertici i punti medi del triangolo che lo contiene. Questa scomposizione porta facilmente al seguente algoritmo iterativo 8.

**Algoritmo 8** - $triangoli(l, n)$ : sequenza di $n$ triangoli di lato $l$

**Input:** lunghezza $l$ del lato del triangolo esterno, numero $n$ di triangoli

1: **for** $n$ **times do**
2: disegna un triangolo di lato $l$
3: spostati sul punto medio di un lato del triangolo appena disegnato
4: orientati verso un altro punto medio del triangolo appena disegnato
5: dimezza la lunghezza $l$ del lato
6: **end for**

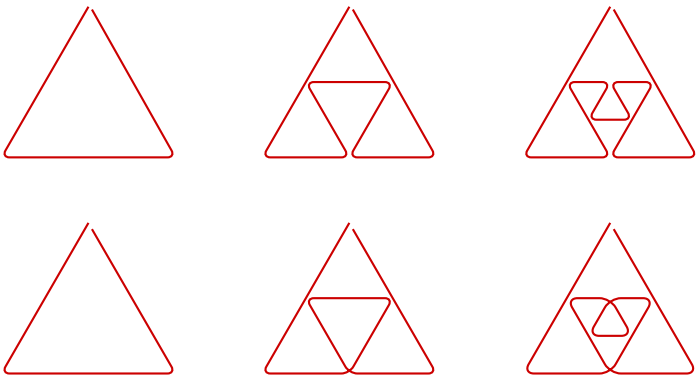
La figura sopra descritta può essere vista, adottando un'ottica ricorsiva, formata da un triangolo esterno in cui è iscritta, appoggiata sui punti medi del triangolo, una figura similare, composta da un triangolo in meno. Questa impostazione porta direttamente al seguente algoritmo ricorsivo 9.

**Algoritmo 9** - $triangolazione(l, n)$ : triangolazione di lato $l$ ed ordine $n$

**Input:** lunghezza $l$ del lato del triangolo esterno, ordine $n$ della figura

1: disegna un triangolo di lato $l$
2: **if** $n > 1$ **then**
3: spostati sul punto medio di un lato del triangolo appena disegnato
4: orientati verso un altro punto medio del triangolo appena disegnato
5: $triangolazione(l/2, n-1)$
6: **end if**

L'algoritmo 9 risulta poco efficiente perché (alla linea 3.) comporta uno spostamento improduttivo da un vertice del triangolo appena disegnato al punto medio di uno dei suoi tre lati appena disegnati. Per migliorare questo aspetto disegnamo la figura senza passare su un tratto già disegnato, percorrendo un circuito hamiltoniamo, ossia senza alzare la penna dal foglio e senza passare su un tratto già disegnato. Questa possibilità è garantita dal fatto che si tratta di un grafo avente tutti i suoi nodi di ordine 4 che è pari; e questo garantisce, per una proprietà dei grafi, che sia possibile disegnare la figura senza alzare la matita dal foglio e senza passare su un tratto già disegnato. Le figure che seguono descrivono due possibili soluzioni di circuito di disegno.



Le due soluzioni suggerite dalle due sequenze di figure sopra descritte risultano equivalenti come risultato finale, ma, al fine di descriverle algoritmicamente, la prima presenta la difficoltà che le sottofigure autosimili devono essere alternando il verso di percorrenza. La seconda risulta più facilmente descrivibile, come riportato nell'algoritmo ricorsivo 10.

**Algoritmo 10** - $circuitotriangoli(l, n)$ : circuito hamiltoniano di triangoli

**Input:** lunghezza $l$ del lato del triangolo esterno, ordine $n$ della figura

1: $forward(l)$
2: $left(120)$
3: $forward(l/2)$
4: $left(60)$
5: **if** $n > 1$ **then**
6: $\quad circuitotriangoli(l/2, n-1)$
7: **end if**
8: $right(60)$
9: $forward(l/2)$
10: $left(120)$
11: $forward(l)$
12: $left(120)$

## 4.6 La tartaruga si muove nel piano

Una significativa evoluzione della grafica della tartaruga consiste nel denotare con dei nomi identificativi i punti di stazionamento della tartaruga, in modo da poterli utilizzare nel seguito del programma, usando i nomi precedentemente definiti. Questa possibilità si fonda sulla funzione *pos* che fornisce il punto in cui staziona la tartaruga. Per denotare con un nome, ad esempio $P$, il punto di stazionamento attuale della tartaruga viene utilizzata l'istruzione di assegnazione

$$P \leftarrow pos()$$

Similmente a *pos* si può usare la funzione *dpos* che ritorna il punto di stazionamento della tartaruga, con la differenza che un'assegnazione della forma $P \leftarrow dpos()$ comporta che il punto $P$ viene automaticamente modificato qualora la tartaruga si muova. Avendo, in questo modo, la possibilità di denotare con dei nomi i punti del piano, risultano coerenti e ben fondate le seguenti istruzioni:

| | |
|---|---|
| $move(P)$ | muove la tartaruga sul punto $P$, senza disegnare |
| $draw(P)$ | sposta la tartaruga sul punto $P$, disegnando una linea |
| $look(P)$ | orienta la tartaruga verso il punto $P$, senza spostarla |
| $dist(P)$ | distanza della tartaruga dal punto $P$ |

Le primitive *move* e *draw* non modificano l'angolo di avanzamento della tartaruga e lo stato su/giù della penna.

*Esempio 4.6.1* - Il disegno di un triangolo di dati vertici può essere realizzato mediante il seguente algoritmo.

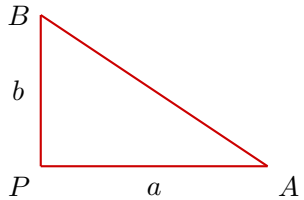
**Algoritmo 11** - $triangolo(A, B, C)$ - triangolo di vertici $A, B, C$

**Input:** vertici $A, B, C$ del triangolo

1: $move(A)$
2: $draw(B)$
3: $draw(C)$
4: $draw(A)$

*Problema 4.6.1* Disegnare un triangolo rettangolo di dati cateti.

*Soluzione*. Il disegno di un triangolo rettangolo di cateti $a$ e $b$ ed avente il vertice dell'angolo retto sul punto attuale di stazionamento della tartaruga può essere realizzato mediante il seguente algoritmo 12.



**Algoritmo 12** - *triangoloRettangolo*$(a, b)$ - triangolo rettangolo di cateti $a, b$

**Input:** lunghezze $a$ e $b$ dei cateti
1: $P \leftarrow pos()$
2: $forward(a)$
3: $A \leftarrow pos()$
4: $move(P)$
5: $right(90)$
6: $forward(b)$
7: $draw(A)$

□

Insegnando alla tartaruga delle nuove istruzioni è possibile realizzare un livello di funzionalità in cui la tartaruga non viene più percepita come tale, ma diventa uno strumento invisibile che lascia trasparire delle funzionalità puramente geometriche. Ciò è confermato dai seguenti algoritmi che evidenziano che la tartaruga può essere usata come *strumento di disegno* (algoritmo 13), *strumento di misura* (algoritmo 14) e *strumento di calcolo di punti* (algoritmo 15).

**Algoritmo 13** - *segmento*$(A, B)$ - disegno del segmento fra due punti $A$ e $B$

**Input:** punti $A$ e $B$
1: $move(A)$
2: $draw(B)$

**Algoritmo 14** - *distanza*$(A, B)$ - distanza fra due punti $A$ e $B$

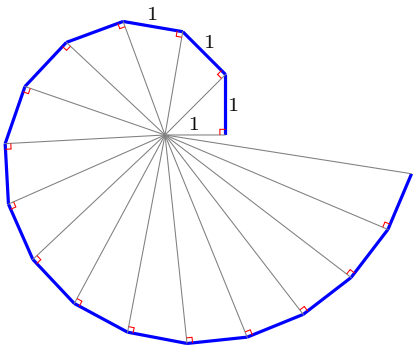
**Input:** punti $A$ e $B$
1: $move(A)$
2: **return** $dist(B)$

**Algoritmo 15** - *medio*$(A, B)$ - punto medio fra i due punti $A$ e $B$

**Input:** punti $A$ e $B$
1: $move(A)$
2: $look(B)$
3: $jump(dist(B)/2)$
4: **return** $pos()$

*Osservazione*. Una tipica estensione dell'ambiente del piano geometrico consiste nel considerare un sistema di riferimento cartesiano mediante il quale un punto risulta individuato dalla coppia numerica delle sue coordinate; in questo modo si permette l'uso del calcolo algebrico a supporto della soluzione di problemi geometrici nel piano.

*Problema 4.6.2* La *spirale di Teodoro*, così chiamata perché tracciata per la prima volta da Teodoro di Cirene (V secolo a.C.), viene costruita partendo da un cateto di un triangolo rettangolo isoscele di cateti di lunghezza unitaria; i successivi lati della spirale sono costituiti dai cateti di lunghezza unitaria dei triangoli aventi come altro cateto l'ipotenusa del triangolo precedente, come mostrato nella figura che segue. Disegnare la spirale.




*Soluzione*. La costruzione della spirale si fonda sulla memorizzazione del punto di posizionamento iniziale che costituirà il centro della spirale; il procedimento completo è descritto nell'algoritmo 16.

**Algoritmo 16** - $teodoro(n)$ : spirale di Teodoro con $n$ lati

**Input:** numero $n$ di lati della spirale

1: $P \leftarrow pos()$ ▷ posizione iniziale della tartaruga
2: $jump(1)$
3: **for** $n$ **times**
4: $look(P)$
5: $right(90)$
6: $forward(1)$
7: **end for**

□

*Nota.* La spirale di Teodoro gode di molte interessanti proprietà geometriche. Ad esempio, estendendo indefinitamente la spirale non ci saranno ipotenuse che si sovrappongono; di più, le rette di sostegno dei lati dei triangoli sono tutte distinte.

## 4.7 La tartaruga come automa con stato

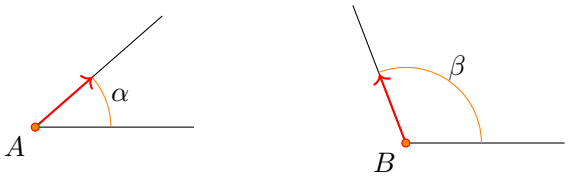
La tartaruga è un *automa con stato*; lo stato è costituito dai seguenti attributi:

- posizione $P$ attuale
- angolo $\alpha$ di avanzamento
- stato $s$ (su/giù) della penna

Oltre a questi attributi, di tipo geometrico, se ne possono considerare altri, di tipo grafico, quali: colore della penna, spessore della linea, ed altri.

Lo stato della tartaruga può essere modificato agendo direttamente su di essa, mediante dei comandi. Gli attributi dello stato della tartaruga sono accessibili tramite le funzioni *pos*, *getang* ed *ispendown*.

Lo stato della tartaruga che si trova sul punto $A$, con angolo di avanzamento pari ad $\alpha$ e con la penna nello stato $s$ è definito mediante una terna del tipo $(A, \alpha, s)$. Per passare dallo stato $(A, \alpha, s)$ allo stato $(B, \beta, t)$ si può eseguire l'algoritmo 17.




**Algoritmo 17** - Passaggio dallo stato $(A, \alpha, s)$ allo stato $(B, \beta, t)$

```
1: move(B)
2: setang(β)
3: if s ≠ t then
4:    if s then
5:       pendown()
6:    else
7:       penup()
8:    end if
9: end if
```

Lo *stato* (posizione, angolo, stato (su/giù) della penna) attuale della tartaruga costituisce una sorta di memoria attuale, istantanea, che serve per gestire il comportamento della tartaruga nell'esecuzione di ciascun comando. Un'evoluzione di questa situazione consiste nel memorizzare in una struttura di memoria la storia di alcuni istanti passati, ciascuno dei quali descritto mediante un'immagine dello stato. Una semplice struttura di memoria adatta per salvare e ripristinare degli stati è costituita da una pila in cui vengono salvati, uno sopra l'altro, i vari stati. Gli stati memorizzati possono essere successivamente ripresi e ristabiliti, a partire dai più recenti ai più vecchi, mediante i

seguenti comandi che memorizzano e recuperano le informazioni in una pila di stati:

| | |
|---|---|
| *init*() | riporta la tartaruga allo stato iniziale |
| *push*() | salva nella pila lo stato attuale della tartaruga |
| *pop*() | ripristina come attuale l'ultimo stato salvato |

Ogni comando ha l'effetto di modificare lo stato interno della tartaruga ed eventualmente di produrre un effetto grafico. Come in ogni sistema con memoria, cambiamento di stato ed effetto prodotto dipendono dal comando e dallo stato precedente. La situazione è descritta nella figura 4.7.

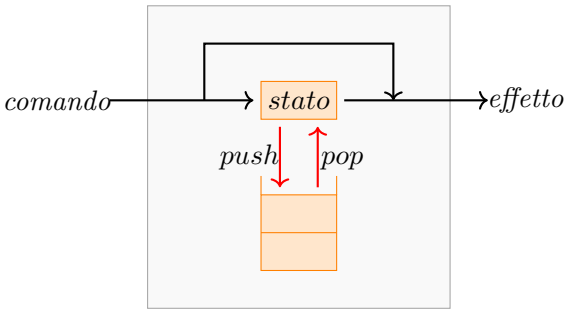
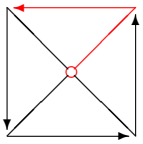



Figura 4.7: Architettura interna dell'automa della grafica della tartaruga.

*Problema 4.7.1* Disegnare la seguente figura:

*Soluzione*. Una soluzione consiste nel percorrere il perimetro del quadrato, memorizzando, strada facendo, in 4 variabili distinte i 4 vertici del quadrato e disegnare le due diagonali congiundendo a coppie i vertici. Questa strategia ha il difetto complicare, se non precludere, un efficace uso di un ciclo e una semplice generalizzazione ad un generico poligono regolare. Una soluzione alternativa consiste nel considerare la figura scomposto come descritta a seguire, dove è evidenziato in rosso l'elemento che si ripete e con un circoletto il punto di partenza della tartaruga:



L'algoritmo che ne deriva è il seguente:

**Algoritmo 18** - Disegno di un quadrato con diagonali

**Input:** lunghezza $l$ della semidiagonale

```
1: for 4 times
2:     salva lo stato corrente
3:     disegna un elemento
4:     ripristina l'ultimo stato salvato
5:     ruota di 90 gradi
6: end for
```

L'algoritmo precedente può essere dettagliato come descritto nell'algoritmo 19.

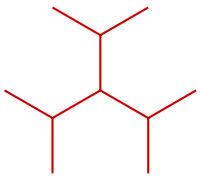
**Algoritmo 19** - Disegno di un quadrato con diagonali

**Input:** lunghezza $l$ della semidiagonale

```
1: for 4 times
2:     push()
3:         push()
4:             jump(l)
5:             P ← pos()
6:         pop()
7:         right(90)
8:         forward(l)
9:         draw(P)
10:    pop()
11:    right(90)
12: end for
```

Nell'algoritmo 19 sono state evidenziate mediante indentazione le sequenze di azioni comprese fra la memorizzazione dello stato nello stack (*push*) ed il corrispondente ripristino dallo stack (*pop*). □

*Problema 4.7.2* Disegnare un *trifolio* come riportato nella figura che segue.



*Soluzione*. Guidati da un principio di simmetria, conviene assumere il punto centrale del trifoglio come punto di partenza della tartaruga; in questo modo

risulta più facile descrivere un algoritmo iterativo (algoritmo 18) che fa ricorso alle operazioni *push* e *pop* per il salvataggio dello stato.

**Algoritmo 20** - *trifolio*($l$) : trifolio di lato di lunghezza $l$

**Input:** lunghezza $l$ della semidiagonale

```
1:  for 3 times
2:      push()
3:          forward(l)
4:          push()
5:              left(60)
6:              forward(l)
7:          pop()
8:          right(60)
9:          forward(l)
10:     pop()
11:     right(120)
12: end for
```

Nell'algoritmo 20 si può notare che le operazioni *push* e *pop* compaiono come delle parentesi virtuali che racchiudono i comandi di movimento. Indicando con { e } queste parentesi, l'algoritmo può essere scritto in forma compatta, come segue:

$$3 * [\{forward(l), \{left(60), forward(l)\}, right(60), forward(l)\}, right(120)]$$

□

*Problema 4.7.3* Determinare il $k$-esimo vertice $P_k$ di un poligono regolare di $n$ lati, percorrendo il poligono in senso antiorario. Sono dati due punti $P_1$ e $P_2$ consecutivi del poligono, il numero $n$ dei suoi vertici e l'indice $k$ del punto da determinarsi.

*Soluzione*. Il procedimento è descritto nell'algoritmo 21.

**Algoritmo 21** - vertice $P_k$

**Input:** punti $P_1$ e $P_2$ iniziali, numero $n$ dei vertici, ordine $k$ del vertice

```
1: move(P_1)
2: look(P_2)
3: l ← dist(P_2)
4: repeat k − 1 times
5:     jump(l)
6:     left(360/n)
7: end repeat
8: return pos()
```

□

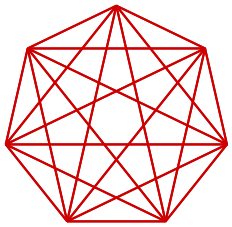
## 4.8 Disegno di un grafo completo

Arrivati a questo punto abbiamo a disposizione tre modalità operative diverse della grafica della tartaruga:

1. tartaruga elementare, con le sole operazioni *forward*, *left*, *right*, con possibilità di gestire la penna (funzioni *penup*, *pendown*)
2. tartaruga con possibilità di denotazione dei punti (funzioni *pos*, *move*, *draw*)
3. tartaruga come automa con stato, con possibilità di salvare lo stato e di ripristinarlo (funzioni *push*, *pop*)

Il problema che segue ingloba tutte le modalità operative della tartaruga precedentemente esaminate.

*Problema 4.8.1* Un *grafo completo* è la figura che si ottiene unendo con dei segmenti i vertici di un poligono regolare in tutti i modi possibili. La figura che segue descrive un grafo completo di 7 vertici. Disegnare un grafo completo.



*Soluzione*. Indicando con $P_1$, $P_2$, ..., $P_n$ i vertici del poligono, il grafo può essere disegnato basandosi sull'algoritmo 22.

**Algoritmo 22** - *grafo*$(n,l)$ : grafo completo - ver.1

**Input:** numero $n$ dei vertici, lunghezza $l$ del lato

```
1: for i from 1 to n − 1 do
2:     for j from i + 1 to n do
3:         congiungi il vertice P_i con il vertice P_j
4:     end for
5: end for
```

A questo punto rimane da risolvere il problema di determinare un generico punto del poligono; questo problema può essere risolto con approcci diversi. Basandosi sulle operazioni fondamentali della tartaruga i punti del poligono possono essere determinati come segue: il punto $P_1$ viene inizializzato con la posizione iniziale della tartaruga (linea 2 dell'algoritmo 23); i successivi punti $P_i$ vengono determinati con un'azione di avanzamento e rotazione (linee 12–13 dell'algoritmo 23); i punti $P_j$ vengono calcolati con un ciclo di $j - i - 1$ azioni

di avanzamento e rotazione, a partire da $P_i$ (linee 5 – 8 dell'algoritmo 23). In questo modo si arriva al definitivo algoritmo 23.

**Algoritmo 23** - *grafo*$(n, l)$ : grafo completo - ver.2

**Input:** numero $n$ dei vertici, lunghezza $l$ del lato

```
 1: for i from 1 to n − 1 do
 2:     P_i ← pos()
 3:     a ← getang()
 4:     for j from i + 1 to n do
 5:         for j − i times
 6:             jump(l)
 7:             left(360/n)
 8:         endfor
 9:         draw(P_i)
10:         setang(a)
11:     end for
12:     jump(l)
13:     left(360/n)
14: end for
```

Usando la tartaruga come automa con stato ripristinabile, il grafo può essere disegnato mediante l'algoritmo 24.

**Algoritmo 24** - *grafo*$(n, l)$ : grafo completo di $n$ vertici e lato di lunghezza $l$

**Input:** numero $n$ dei vertici, lunghezza $l$ del lato

```
 1: α ← 360/n
 2: for i from 1 to n − 1 do
 3:     P ← pos()
 4:     jump(l)
 5:     left(α)
 6:     push()
 7:     for j from i + 1 to n do
 8:         push()
 9:         draw(P)
10:         pop()
11:         jump(l)
12:         left(α)
13:     end for
14:     pop()
15: end for
```

Una soluzione alternativa consiste nel far fare alla tartaruga un giro periferico sul bordo del poligono, memorizzando in un array i vertici del poligono; dopodicché con due cicli *for* annidati si disegna facilmente il grafo completo. □

## 4.9 La tartaruga come calcolatore

La funzione $dist(P)$, che ritorna la distanza della tartaruga da un generico punto $P$ del piano, permette di utilizzare la tartaruga come un rudimentale strumento di calcolo, combinando le sue capacità di spostamento, rotazione e misurazione di distanza da un punto. La tecnica consiste nell'allontanare la tartaruga dal punto iniziale, seguendo un percorso descritto dall'algoritmo; alla fine si ottiene il risultato misurando la distanza della tartaruga dal punto iniziale. Per muovere la tartaruga senza vedere effetti collaterali grafici è sufficiente muovere la tartaruga con la penna alzata oppure eseguire gli spostamenti mediante il comando *jump*.

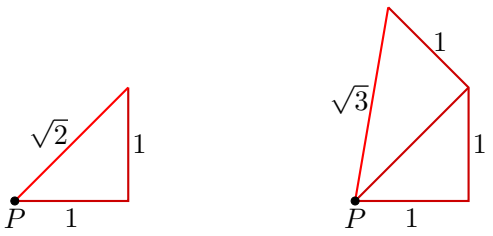
*Esempio 4.9.1 -* Il calcolo di $\sqrt{2}$ può essere svolto mediante il seguente algoritmo 25 che si basa sul teorema di Pitagora applicato ad un triangolo rettangolo di cateti di lunghezza unitaria.

**Algoritmo 25** - Calcolo di $\sqrt{2}$

**Input:** nessun dato
**Output:** valore di $\sqrt{2}$

1: $P \leftarrow pos()$
2: $jump(1)$
3: $left(90)$
4: $jump(1)$
5: **return** $dist(P)$

In modo analogo si può calcolare la $\sqrt{3}$, come descritto nella figura che segue.




Il procedimento descritto nell'algoritmo 25 può essere generalizzato per calcolare la radice quadrata di un generico numero naturale, basandosi sulla costruzione della spirale di Teodoro descritta nel problema 4.6.2. Si lascia per esercizio la stesura dell'algoritmo.

## 4.10 La grafica della tartaruga in modalità ad oggetti

Seguendo la tradizione, come fatto nei precedenti paragrafi, la geometria della tartaruga viene presentata secondo un'impostazione *imperativa*: esiste una tartaruga alla quale si inoltrano i comandi che, eseguiti, producono un effetto su video. Una variante di questa impostazione è fornita dalla modalità *orientata agli oggetti*: anziché rivolgere i comandi *alla* tartaruga, vengono create delle tartarughe, ciascuna avente un proprio nome, a ciascuna delle quali si può inoltrare un comando, l'esecuzione del quale avrà un effetto che dipende dal particolare stato della tartaruga alla quale è stato rivolto.

La grafica della tartaruga impostata secondo l'impostazione orientata agli oggetti si fonda sui seguenti due aspetti sintattici:

- per creare una tartaruga $t$ con specifici argomenti di inizializzazione viene richiamato il costruttore della classe ed utilizzata un'assegnazione:

$$t \leftarrow Turtle()$$

- per eseguire un'*azione* [1] (comando o funzione) viene utilizzata la notazione puntata

$$t.azione(\dots)$$

*Esempio 4.10.1 -* La seguente porzione di algoritmo disegna un quadrato di lato di 10 unità:

```
1: t ← Turtle()
2: for 4 times
3:     t.forward(10)
4:     t.left(90)
5: end for
```

Adottando l'impostazione orientata agli oggetti risulta semplice disegnare delle figure composte da parti, ciascuna delle quali viene disegnata da una diversa tartaruga. L'effetto grafico risulta dinamicamente interessante se le diverse tartarughe si muovono contemporaneamente; questo effetto può essere simulato associando un *thread* di esecuzione a ciascuna tartaruga oppure con un ciclo in cui si fanno eseguire brevi spostamenti a tutte le tartarughe. L'impostazione orientata agli oggetti permette anche di simulare l'interazione fra tartarughe diverse.

*Esempio 4.10.2 -* Nella porzione di algoritmo che segue viene creata una sequenza di tartarughe che vengono inizialmente orientate verso diverse direzioni, a raggiera; successivamente vengono fatte avanzare ottenendo l'effetto di una spirale a più rami di diversi colori che si sviluppa dal centro verso l'esterno.

[1] Nella terminologia orientata agli oggetti un'azione viene detta *metodo*.

```
1: n ← 17        ▷ numero di tartarughe
2: t ← n ∗ [None]
3: colors ← [RED, GREEN, BLUE]
4: for k from 0 to n − 1 do
5:     t[k] ← Turtle()
6:     t[k].color(colors[k mod len(colors)])
7:     t[k].left(360/n ∗ k)
8: end for
9: for 100 times
10:    for k from 0 to n − 1 do
11:        t[k].forward(.1)        ▷ passi di lunghezza 1
12:        t[k].left(2)            ▷ rotazione di 2^o
13:    end for
14: end for
```

*Osservazione*. L'impostazione orientata agli oggetti risulta particolarmente produttiva in quanto, fra tante altre interessanti caratteristiche, permette di realizzare in modo semplice e veloce delle nuove classi di oggetti, *estendendo* classi già realizzate, aggiungendo nuovi metodi oppure ridefinendo il comportamento di metodi già realizzati, *ereditando* tutti i metodi già presenti nella classe che si estende. Ad esempio è possibile realizzare una nuova classe *Tarta*, estendendo la classe *Turtle*, aggiungendo nuovi metodi per gestire la velocità di avanzamento delle tartarughe ed ereditando tutti i metodi già presenti nella classe *Turtle*.

*Osservazione*. L'impostazione orientata agli oggetti permette di gestire più stati della tartaruga senza ricorrere a salvare e ripristinare lo stato della tartaruga, in quanto ciascuna tartaruga incapsula il proprio stato.

## ESERCIZI

4.1 Con riferimento alla geometria della tartaruga, discutere quali delle seguenti equivalenze sono vere:

| | | |
|---|---|---|
| $forward(x+y)$ | equivale a | $forward(x)$, $forward(y)$ |
| $left(a+b)$ | equivale a | $left(a)$, $left(b)$ |
| $left(a-b)$ | equivale a | $left(a)$, $right(b)$ |
| $left(360+a)$ | equivale a | $left(a)$ |
| $left(360-a)$ | equivale a | $right(a)$ |
| $left(180+a)$ | equivale a | $right(180-a)$ |
| $left(180-a)$ | equivale a | $right(180+a)$ |

4.2 Dimostrare che per disegnare un poligono regolare la tartaruga compie delle rotazioni che sommate danno un angolo giro. Dimostrare che questo risultato vale per disegnare un generico poligono (anche non regolare) convesso. Disegnare un poligono regolare di $n$ lati di lunghezza $l$.

4.3 Disegnare un rettangolo di lati di lunghezze $a, b$.

4.4 Disegnare un triangolo isoscele di base $b$ ed altezza $h$.

4.5 Disegnare un triangolo di lati di lunghezze $a, b, c$.

4.6 Determinare, senza eseguire al computer, le figure disegnate mediante le seguenti porzioni di algoritmi:

1.

```
1: for 8 times
2:     forward(l)
3:     left(45)
4:     forward(l)
5: end for
```

2.

```
1: for 8 times
2:     forward(l)
3:     for 4 times
4:         left(45)
5:         forward(l)
6:     end for
7: end for
```

4.7 Determinare, senza eseguire al computer, le figure disegnate mediante le seguenti porzioni di algoritmi:

1.

```
1: for 8 times
2:     forward(l)
3:     push()
4:         forward(l)
5:     pop()
6:     left(45)
7: end for
```

2.

```
1: for 8 times
2:     forward(l)
3:     push()
4:         left(45)
5:         forward(l)
6:     pop
7: end for
```

4.8 Mediante la grafica della tartaruga, un poligono regolare può essere disegnato equivalentemente mediante rotazioni a sinistra o a destra. Stabilire la relazione (uguaglianza, equivalenza, isometria, similitudine, simmetria, ...) che esiste fra due poligoni disegnati l'uno con rotazioni a sinistra, l'altro con rotazioni a destra, partendo, nei due casi, dalla stessa posizione ed orientamento iniziale della tartaruga.

4.9 Sia $F$ una sequenza di espressioni per disegnare una figura. Stabilire quale relazione (uguaglianza, equivalenza, isometria, similitudine, simmetria, ...) intercorre fra la figura $F$ e le seguenti:

$F_1$: *jump*$(x)$, $F$

$F_2$: *left*$(a)$, $F$

$F_3$: *jump*$(x)$, *left*$(a)$, $F$

$F_4$: *left*$(a)$, *jump*$(x)$, $F$

4.10 Sia $F(\alpha)$ una figura definita mediante la geometria della tartaruga, con angoli di rotazione di ampiezza $f(\alpha)$, essendo $f$ una funzione lineare di $\alpha$. Stabilire le relazioni esistenti fra le seguenti figure: $F(\alpha)$, $F(\alpha+90)$, $F(\alpha+180)$, $F(\alpha+360)$, $F(\alpha/2)$.

4.11 Sia $F(x)$ una figura composta da segmenti aventi lunghezze proporzionali a $x$. Stabilire la relazione che intercorre tra la figura $F(x)$ e la figura $F(k\,x)$, essendo $k$ un parametro reale positivo.

4.12 Disegnare una figura formata da una serie di quadrati concentrici, coassiali ed equispaziati. Calcolare la lunghezza complessiva dei tratti costituenti la figura.

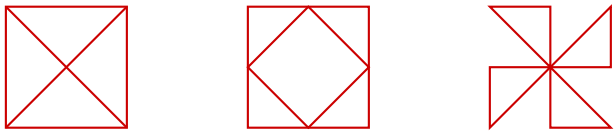
4.13 Disegnare una figura costituita da una sequenza di triangoli equilateri ciascuno avente per vertici i punti medi del triangolo nel quale è inscritto. Calcolare la lunghezza complessiva dei tratti costituenti la figura.

4.14 Disegnare una spirale avente lo stesso angolo di rotazione fra i lati ed avente i lati le cui misure costituiscono una progressione aritmetica. Calcolare la lunghezza complessiva dei tratti costituenti la figura.

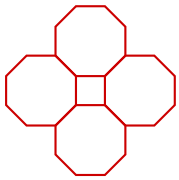
4.15 Disegnare una $(n-k)$-stella avente i vertici su $n$-poligono regolare di lato di lunghezza $l$, avente i lati costruiti saltando di $k$ vertici sul poligono; per $k = 1$ si ottiene un poligono regolare. Analizzare per quali valori di $n$ e $k$ la stella passa per tutti i vertici del poligono con un solo percorso ciclico.

4.16 Determinare un generico punto $P_k$ di un poligono regolare di $n$ lati, percorrendo il poligono in senso antiorario. Sono dati due punti $P_1$ e $P_2$ consecutivi del poligono, il numero $n$ dei suoi vertici e l'indice $k$ del punto da determinarsi.

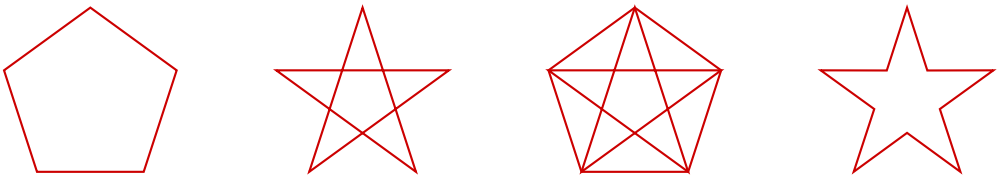
4.17 Disegnare le seguenti figure. Si richiede di usare negli spostamenti della tartaruga solo numeri naturali o razionali.



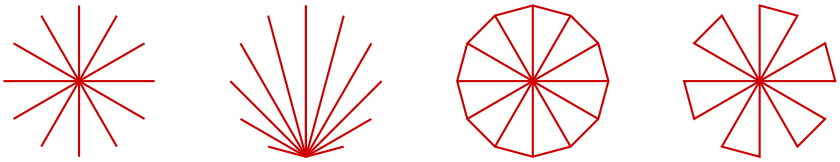
4.18 Disegnare la seguente figura.



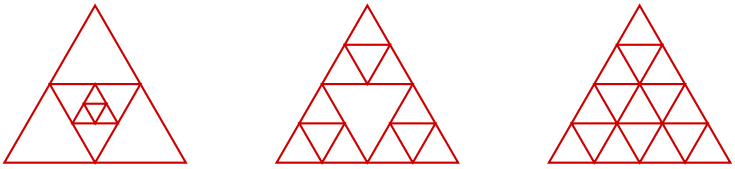
4.19 Disegnare le seguenti figure composte da segmenti che uniscono i vertici di un pentagono regolare. Generalizzare al caso di un poligono di $n$ vertici.



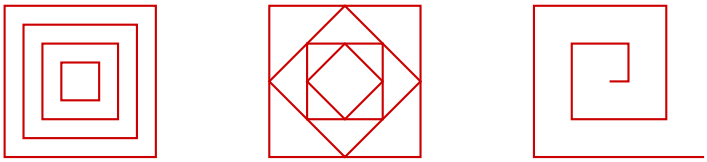
4.20 Disegnare le seguenti figure composte da raggi che uniscono i vertici di un poligono regolare.



4.21 Disegnare, in modalità iterativa ed in modalità ricorsiva, le seguenti figure composte da una sequenza di triangoli equilateri inscritti ciascuno in un altro ed aventi per vertici i punti medi del triangolo immediatamente precedente. La prima figura replica il procedimento solo sul triangolo centrale, la seconda su ciascun triangolo agli angoli e la terza su tutti i triangoli.

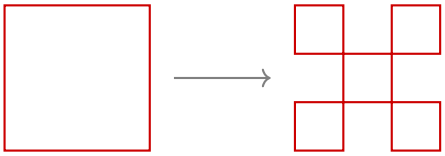


4.22 Disegnare, in modalità iterativa ed in modalità ricorsiva, le seguenti figure. Determinare la lunghezza di ciascuna figura (mediante un algoritmo o mediante una formula chiusa).



4.23 Generare un'istanza della successione di figure generate dalla seguente regola ricorsiva:

- una figura di ordine 0 e lato $h$ è costituita da un quadrato di lato $h$
- una figura di ordine $n$ viene generata suddividendo ciascun quadrato della figura di ordine $n-1$ in 9 quadrati uguali, eliminando i tratti centrali di ciascun lato

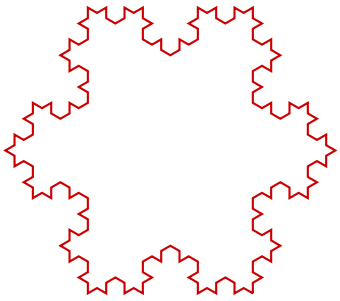


4.24 Una *tassellazione* del piano consiste nella partizione del piano mediante un disegno di figure appartenenti ad una classe predefinita, similmente a come si ricopre un pavimento mediante delle piastrelle di forme prefissate. Definire delle funzioni per generare una tassellazione del piano (una porzione del video) definita mediante le seguenti regole:

- la tassellazione è costituita da quadrati ed ottagoni regolari di uguali lati
- ogni quadrato è circondato da 4 ottagoni (ciascuno su ogni lato del quadrato)
- ogni ottagono è circondato da 4 quadrati (ciascuno alternato sui lati dell'ottagono)

4.25 Una figura è composta da una sequenza di triangoli uno dentro l'altro, ciascuno avente per vertici i punti medi del triangolo esterno contiguo, secondo il seguente schema ricorsivo:
$T_0$ : triangolo assegnato
$T_{n+1}$ : è il triangolo ottenuto da $T_n$ congiungendone i punti medi dei lati
Definire in forma iterativa ed in forma ricorsiva una funzione che generi la figura sopra descritta.
Determinare il punto a cui tende la successione dei triangoli descritta sopra. Discutere la convergenza del procedimento iterativo.

4.26 Un *fiocco di von Kock* è ottenuto sostituendo i lati di un triangolo equilatero con tre curve di von Kock. Determinare, in funzione della lunghezza del lato di base e dell'ordine della curva, il perimetro e l'area di un fiocco di von Kock. Valutare questi valori al tendere all'infinito dell'ordine della curva. Disegnare un fiocco di von Kock.



4.27 Determinare la lunghezza di un albero binario $T(l, n)$.

4.28 La ricorsività unita alla generazione di numeri casuali può costituire la base per ottenere dei disegni e degli effetti grafici gradevoli, molto utilizzati in alcuni campi della *computer graphics*. In particolare gli alberi binari possono essere forme più simili agli alberi botanici agendo sui seguenti aspetti:

- definire un parametro $k$ che esprime il rapporto fra la lunghezza di un ramo e la lunghezza del suo ramo padre
- definire lo spessore della linea che man mano si assotiglia andando verso le foglie
- aggiungere delle foglie alle estremità dell'albero
- aggiungere delle ramificazioni lungo i fusti
- aggiungere più diramazioni di ciascun ramo
- ricorrere a dei numeri casuali che intervengono in tutti i precedenti aspetti

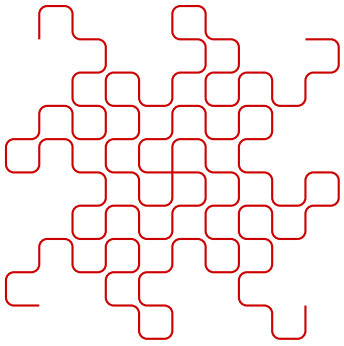
Definire una funzione ricorsiva per generare un *albero casuale*. Tali figure vengono dette *piante graftali*.

4.29 Determinare il numero di segmenti che compongono una curva del drago di ordine $n$.

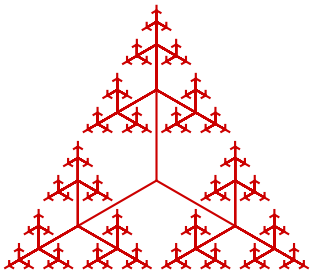
4.30 Determinare la lunghezza di una curva del drago di ordine $n$ costruita a partire da un segmento iniziale di lunghezza $l$ (curva di ordine 0).

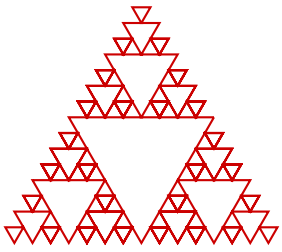
4.31 Nonostante l'apparenza, la curva del drago è costituita da una linea continua che non interseca mai se stessa. Definire una funzione che generi una curva del drago in modo da smussare gli spigoli per evidenziare meglio la struttura lineare della curva.

4.32 Combinando 4 istanze della curva del drago si può ottenere una figure più articolata, come descritto nella seguente figura. Generare queste figure.

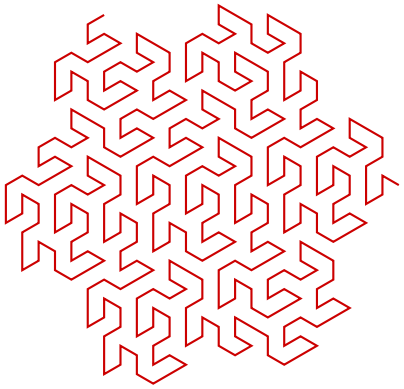


4.33 Individuare gli schemi ricorsivi che generano le seguenti figure. Generare le figure.





4.34 Individuare lo schema ricorsivo che genera la seguente figura, detta curva di Peano-Gosper. Generare la figura.



4.35 Disegnare la spirale di Teodoro con i raggi congiungenti ciascun vertice con il centro della spirale.

4.36 Dimostrare che i raggi della spirale di Teodoro hanno lunghezze pari a $\sqrt{1}$, $\sqrt{2}$, $\sqrt{3}$, $\sqrt{4}$, $\sqrt{5}$, ..., $\sqrt{n}$, ...

4.37 Determinare l'ampiezza dell'angolo opposto all'$n$-esimo lato della spirale di Teodoro.

4.38 Dimostrare che la distanza tra due avvolgimenti della spirale di Teodoro, con il tendere all'infinito del numero di giri, la distanza tra due avvolgimenti consecutivi tende (rapidamente) a $\pi$.

4.39 Usando la tartaruga, determinare la lunghezza di un poligono intrecciato costituito da una stella a 5 punte congiungente i vertici di un pentagono regolare di lato 100.

4.40 Usando la tartaruga calcolare il valore assoluto della differenza (distanza) fra due dati numeri.

4.41 Usando la tartaruga, dati tre punti $A$, $B$, $C$, determinare l'ampiezza dell'angolo $A\widehat{B}C$.

4.42 Usando la tartaruga determinare il perimetro di un triangolo avente due lati di lunghezza $a$ e $b$ e l'angolo fra essi compreso di ampiezza $\alpha$.

4.43 Determinare la lunghezza della base di un triangolo isoscele aventi i lati di lunghezza $l$ e l'angolo al vertice di ampiezza $a$.

4.44 Usando la tartaruga come *calcolatore*, calcolare il valore delle seguenti espressioni, essendo $m$ e $n$ numeri naturali:

1. $\sqrt{m}$
2. $\sqrt{m^2+n^2}$
3. $m+n\sqrt{2}$
4. $m+\sqrt{n}$
5. $\sqrt{m}+\sqrt{m}$

4.45 Usando la tartaruga come *calcolatore*, calcolare il *seno* ed il *coseno* di un generico angolo di ampiezza $\alpha$.

4.46 Adottando l'impostazione orientata agli oggetti, simulare il movimento di due tartarughe: una che si muove lungo una circonferenza ed un'altra che la insegue fino a raggiungerla. Simulare il movimento contemporaneo delle due tartarughe mediante un ciclo all'interno del quale le due tartarughe vengono fatte avanzare di piccoli passi e fare delle piccole rotazioni, oppure associando due diversi *thread* di esecuzione a ciascuna delle due tartarughe.

4.47 Posizionare 4 tartarughe ai vertici di un quadrato; ciascuna tartaruga è orientata verso quella posizionata al vertice successivo. Muovere contemporaneamente le 4 tartarughe fino a che si incontrano nel centro del quadrato.

4.48 Alternativamente, in modo più aderente al paradigma *funzionale* [2], si può vedere un comando rivolto alla tartaruga come un'espressione che, elaborata, produce un risultato che, riprodotto su video, produce un effetto grafico. Una figura diventa una struttura dati (composta da alcuni componenti atomici) sulla quale è possibile fare delle operazioni (traslazione, dilatazione, rotazione, ...). Studiare questo approccio alternativo e realizzarlo in un linguaggio di programmazione.

[2]Non si tratta di un'impostazione funzionale pura in quanto il risultato dell'elaborazione dipende dallo *stato* della tartaruga e, d'altra parte, l'elaborazione di un'espressione può modificare lo stato della tartaruga.

# BIBLIOGRAFIA


[1] George A. Bekey, *Autonomous Robots*, The MIT Press, 2005.

[2] Seymour Papert, *The Children's Machine: Rethinking School in the Age of the Computer*, Basic Books, 1993.

[3] Alessandro Bogliolo, *Coding in Your Classroom, Now!*, Giunti Scuola, 2018.

[4] Seymour Papert, *Mindstorms: Bambini computers e creatività*, EMME Edizioni, 1984.

[5] Harold Abelson, Andrea Disessa, *La geometria della tartaruga*, Ed. Franco Muzzio, 1986.

[6] Celia Hoyles, Richard Noss (editors), *Learning Mathematics and Logo*, The MIT Press, 1992.

[7] Paolo Oliva, *Matematica e Logo*, Ed. Franco Angeli, 1993.