LUIGINO CALVI

CODING
LINGUAGGIO
ROBOTICA

2023

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

1

MUOVERE UN ROBOT

Ogni lungo cammino inizia con un primo passo.
Lao Tzu *(filosofo cinese del VI secolo a.C.)*

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 dato problema che richiede un movimento, 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 sotto i diretti comandi impartitigli dall'esterno oppure in base ad un procedimento interno precedentemente programmato 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 l'azione di camminare viene così più correttamente denominata *muovere*.

In questo capitolo viene presentato un semplicissimo automa in grado di muoversi sulle caselle di una scacchiera; viene comandato mediante un altrettanto elementare linguaggio di programmazione dei suoi movimenti.

1.1 Programmabilità dei robot

Un robot può essere programmato in diverse modalità; si possono evidenziare (almeno) i seguenti paradigmi:

- paradigma *strumento*: similmente a come utilizziamo una penna per scrivere o una bicicletta per spostarci, l'uomo aziona direttamente lo strumento in base alla propria volontà e scelta, fornendo i comandi dall'esterno; lo strumento diventa una protesi del corpo dell'uomo e viene controllato dal cervello dell'uomo (la situazione è analoga alla movimentazione di un'automobilina mediante un telecomando o un joistick).
- paradigma *solutore-esecutore*: il solutore (uomo) istruisce l'esecutore su come deve agire e comportarsi; una volta istruito e attivato, l'esecutore agisce autonomamente (in base al programma fornitogli) e l'uomo perde qualsiasi controllo su di esso; il programma deve prevedere la gestione di situazioni non note a priori; è questa la situazione più interessante dal punto di vista algoritmico e per le applicazioni dei robot reali.
- paradigma *auto-apprendimento*: viene precisato l'obiettivo ed il robot cerca autonomamente di raggiungerlo, scegliendo una adeguata strategia di comportamento, imparata autonomamente dalle precedenti esperienze maturate dal robot [1].

I robot eseguono le azioni descritte nel programma, una alla volta, in sequenza. Ci si potrebbe quindi attendere che un programma debba essere composto da una sequenza di comandi e, quindi, da una stringa. Fortunatamente, i linguaggi offrono dei meccanismi descrittivi che consentono al solutore di comprimere i programmi condensandoli in stringhe molto più corte che vengono poi espanse al momento dell'esecuzione. Infatti, un programma può essere definito ricorrendo ad operazioni che combinano porzioni di programma per generare un programma più articolato. La costruzione dei programmi si basa sull'applicazione combinata di pochi schemi di base. Lo schema tipico per definire un linguaggio per controllare un robot si fonda su due passi:

1. definizione delle azioni elementari direttamente eseguibili dal robot; le azioni elementari costituiscono gli atomi mediante i quali realizzare i programmi
2. definizione dei controlli sintattici per organizzare le azioni elementari per costruire i programmi

Questi due passi consentono di dare una definizione ricorsiva di programma come segue: la descrizione di un'azione elementare è un programma; organizzando programmi mediante i controlli si ottengono programmi.

[1] Qui il discorso si complica: tutto ruota attorno al significato che vogliamo attribuire ai termini "imparare autonomamente".

1.2 Muoversi nel piano

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. Nel seguito considereremo una situazione elementare in cui il movimento avviene nel piano.

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. Con queste premesse il rapporto fra le due entità spazio e movente è sintetizzato dal concetto di *stato* che, in una situazione semplificata, unisce l'informazione relativa alla posizione ed alla direzione di avanzamento corrente del movente rispetto allo spazio. Un'entità che si muove autonomamente in uno spazio in base ad un programma viene spesso denotata con il termine *robot*, indipendentemente dalla sua costituzione fisica.

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 non considereremo parametri cinematici quali velocità ed accelerazioni ed il tempo verrà ridotto al concetto di sequenza temporale, in pratica al solo concetto di prima-dopo e, spesso, al solo inizio-fine; 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. Spingendo ancora oltre questa semplificazione si può discretizzare lo spazio e ancor di più limitare lo spazio ad una zona circoscritta. Sarà questo il contesto spaziale che verrà utilizzato nel seguito di questo capitolo.

1.3 Muoversi su una scacchiera

Consideriamo lo spazio a scacchiera: una porzione limitata di piano, suddiviso in un reticolo di 8×8 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 un passo 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*. La situazione è descritta nella figura 1.1.

Il movimento, nella situazione semplificata che stiamo considerando, ha come sue componenti l'avanzamento e la rotazione. In un contesto di spazio discretizzato il movimento può essere ottenuto mediante un avanzamento unitario e la rotazione a sinistra o a destra di un angolo retto; può quindi essere generato combinando pochissime operazioni elementari: avanzamento di un passo e rotazione, unitamente alla possibilità di organizzare queste operazioni in sequenza e mediante la ripetizione.

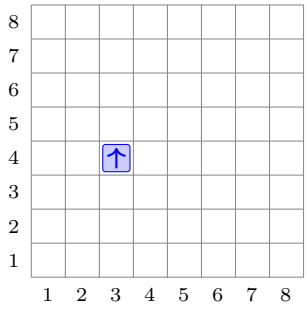


Figura 1.1: Il robot *Quadretto* nel suo ambiente. La freccia denota la direzione di avanzamento.

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. Secondo il paradigma *strumento*, Quadretto può essere movimentato utilizzando specifici bottoni o 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.

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.

1.4 La programmazione

Per muovere Quadretto serve impartirgli dei comandi e per descrivere questi serve un linguaggio. Tecnicamente un *linguaggio* è costituito da un insieme di *parole* o *stringhe* formate da sequenze di caratteri di un prefissato insieme finito e non vuoto di caratteri, detto *alfabeto*. In un contesto di programmazione, secondo il paradigma *solutore-esecutore*, una parola del linguaggio viene generalmente detta *programma*.

Azioni elementari

Quadretto è in grado di eseguire dei movimenti elementari che vengono descritti, rispetto alla propria posizione ed al proprio verso di avanzamento, dall'insieme di valori $\{\texttt{F}, \texttt{B}, \texttt{L}, \texttt{R}\}$ corrispondenti ciascuno ai seguenti movimenti:

- `F` : avanzamento di un passo
- `B` : indietreggiamento di un passo
- `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 θ, che corrisponde a "nessun movimento".

La forma più elementare di programma è costituita da una singola azione elementare.

Esempio 1.4.1 - Ciascuna delle azioni elementari `F`, `B`, `L`, `R` di Quadretto costituisce un programma. □

Un programma può essere costruito combinando un insieme di istruzioni elementari utilizzando alcuni schemi prefissati. Considereremo a seguire tre tipici schemi di programmazione, che consentono di realizzare programmi articolati ed interessanti.

Concatenazione di programmi

Se α e β sono due programmi, mediante la loro *concatenazione*, scritta nella forma $\alpha+\beta$, o semplicemente nella forma $\alpha\beta$, si ottiene il programma costituito dalle azioni di α e di β che verranno eseguite nell'ordine indicato: prima α e poi β. La concatenazione di programmi può essere generalizzata ad una sequenza di programmi α_1, α_2, ..., α_n: la scrittura

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

denota il programma costituito dai programmi indicati, che verranno eseguiti in sequenza, uno dopo l'altro.

Esempio 1.4.2 - La concatenazione del programma `FF` con il programma `FRFF` produce come risultato il programma `FFFRFF`; la sua esecuzione ha l'effetto di far avanzare di 3 passi, ruotare a destra ed avanzare di altri 2 passi. □

Ripetizione di programmi

I robot più semplici, come Quadretto, eseguono le azioni una alla volta, in sequenza. Ci si potrebbe quindi attendere che un programma per questi robot debba essere composto da una sequenza di comandi e, quindi, da una stringa (molto lunga). Fortunatamente, i linguaggi offrono dei meccanismi descrittivi che consentono al solutore di accorciare la scrittura dei programmi condensandoli in stringhe più corte che vengono poi espanse al momento dell'esecuzione.

Coerentemente con la sua natura di indefesso esecutore, un robot può essere programmato ricorrendo a dei controlli che permettono di *ripetere* delle azioni. Se n è un numero naturale ed α è un programma, con

$$n * \alpha$$

oppure semplicemente con $n\,\alpha$, si ottiene il programma costituito dalla concatenazione di n programmi α, ossia $\alpha\alpha\alpha\ldots\alpha$, n volte α.

Esempio 1.4.3 - La scrittura `5F` denota il programma che fa avanzare di 5 passi. □

Raggruppamenti di azioni

Se α_1, α_2, ..., α_n sono programmi, con

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

si denota il programma costituito dalla sequenza dei programmi α_i. L'operazione di raggruppamento risulta necessaria in alcuni contesti, qualora serve considerare un programma come una singola entità, ad esempio per applicare l'operazione di ripetizione.

Esempio 1.4.4 - Raggruppando in sequenza i programmi `F`, `R`, `FF` si ottiene il programma

`[F R FF]`

che equivale al programma `FRFF`. □

Esempio 1.4.5 - La scrittura `[2F L 2F]` denota il programma che fa compiere un percorso ad angolo retto verso sinistra, con entrambi i lati di lunghezza 2 passi. □

Esempio 1.4.6 - La stringa `FFRFFRFFRFFR` muove Quadretto lungo un quadrato di 3 unità di lato, ritornando al punto di partenza. Questa stringa può essere condensata nella stringa `4[2FR]` che genera lo stesso processo della stringa indicata all'inizio. Questo esempio lascia intuire alcuni meccanismi per la compressione dei programmi. □

Inversa di un'azione

Indichiamo con θ l'*azione nulla* ossia l'azione che non comporta alcun cambiamento di stato. 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.

Per ogni azione elementare a l'operazione *inversa* è descritta dalla seguente tabella:

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

Nel seguito, per delle generiche azioni x ed y, useremo anche delle scritture della forma $x - y$ che va intesa come un'abbrevviazione di $x[-y]$ e non come un'operazione binaria − fra x ed y (che non è definita). L'operazione − di inversa è infatti unaria e scritta in notazione prefissa; inoltre, l'operazione inversa − ha priorità sull'operazione di concatenazione; ad esempio, l'azione `-FR` deve essere intesa come $[-$`F`$]$`R`, che equivale a `BR`.

Programmi come espressioni

I meccanismi di *concatenazione*, *ripetizione*, *raggruppamento* ed *inversione* possono essere combinati fra loro, in tutte le combinazioni possibili; si possono ottenere così programmi formati, ad esempio, da "sequenze di ripetizioni di sequenze" e strutture simili.

Le operazioni di raggruppamento [...] ed inversione – hanno priorità massima rispetto alle altre operazioni, mentre l'operazione di ripetizione ha priorità rispetto all'operazione di concatenazione.

Esempio 1.4.7 - Considerando le priorità degli operatori, l'espressione `2[F-RF]` produce il programma `3[FLF]`. □

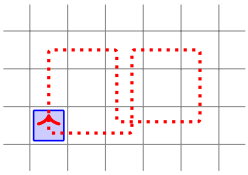
Problema 1.4.1 Percorrere in senso orario il bordo della scacchiera partendo dalla prima casella in basso a sinistra e con la direzione iniziale verso nord.

Soluzione. Il percorso può essere generato mediante la ripetizione di comandi come descritto nel programma `4[7FR]`. □

Problema 1.4.2 Stabilire l'effetto dell'esecuzione del seguente programma:

`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.



□

Ogni problema ha una sua specifica vocazione. Il seguente mira a: verificare la conoscenza della notazione, il concetto di ciclo, la capacità di simulare un procedimento e la capacità di visualizzazione spaziale.

Problema 1.4.3 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 $(3, 2, \texttt{EAST})$. □

1.5 I sottoprogrammi

La maggior parte dei linguaggi di programmazione offre la possibilità di definire *sottoprogrammi*, ossia parti di programma che vengono richiamate ed utilizzate in altre parti del programma. Ogni sottoprogramma viene identificato mediante un nome identificativo. Dal punto di vista operativo la definizione di un sottoprogramma costituisce un comodo artificio per condensare in una parola una parte articolata di programma, favorendo la leggibilità (per il solutore) del programma stesso; dal punto di vista metodologico risulta lo strumento principale per attuare le 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.

Come avviene per qualsiasi sottoprogramma, in tutti i linguaggi di programmazione, la *definizione* permette solo di specificare l'effetto del sottoprogramma, mentre, affinche le azioni vengano svolte si deve *richiamare* il sottoprogramma scrivendo come istruzione il nome identificativo con il quale il sottoprogramma è stato definito.

Esempio 1.5.1 - 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 1.5.2 - La scrittura

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

definisce un sottoprogramma di nome *duepassi* la cui esecuzione fa avanzare Quadretto, nella direzione corrente, di 2 passi. □

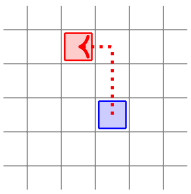
Un programma può essere richiamato all'interno della definizione di un altro sottoprogramma.

Esempio 1.5.3 - Per far fare 2 passi indietro a Quadretto si può definire e successivamente richiamare il seguente sottoprogramma:

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

□

Problema 1.5.1 Muovere Quadretto a mossa di cavallo degli scacchi, come descritto dalla figura che segue.

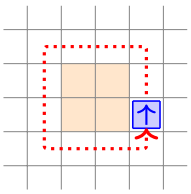


Soluzione. L'immediata soluzione è data dal programma `2FLF`. Per maggiore leggibilità ed in prospettiva di possibili impieghi della soluzione di questo problema, conviene definire il seguente sottoprogramma:

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

□

Problema 1.5.2 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 1.5.1, il percorso a quadrato può essere definito come segue:

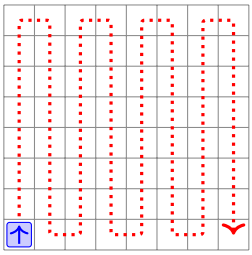
$$quadrato \stackrel{\text{def}}{=} 4 * cavallo$$

È un semplice esempio di applicazione della metodologia bottom up. La metodologia top-down suggerisce naturalmente la seguente alternativa:

1: *lato* ← `3F`	▷ assegnazione
2: *quadrato* ← `4[`*lato*`L]`	▷ uso sottoprogramma *lato*

□

Problema 1.5.3 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 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 assegnazioni che costituiscono un algoritmo che genera il percorso richiesto:

```
1: t ← 7F                    ▷ tratto
2: d ← RFR                   ▷ curva a destra
3: s ← LFL                   ▷ curva a sinistra
4: u ← [t d t]               ▷ percorso a ⊓
5: p ← [u 3[s u]]            ▷ percorso completo
6: return p
```

□

Osservazione. Il linguaggio descritto nei paragrafi precedenti è estremamente semplice ma è povero dal punto di vista della potenza espressiva in quanto non offre molte delle possibilità offerte dai linguaggi di programmazione di tipo professionale (Java, Python, ...). Ad esempio, non permette di

- scrivere espressioni aritmetiche (esempio: `(2+3)F`)
- usare variabili numeriche (esempio: `a=4`)
- esprimere istruzioni di assegnazione fra variabili (esempio: `a=a+1`)
- usare gli operatori di confronto $=, \neq, <, \leq, >, \geq$
- usare i controlli *if*, *while*, ...
- usare sottoprogrammi con argomenti (esempio: `quadrato(5)`)
- usare sottoprogrammi ricorsivi
- usare strutture dati (*insiemi, sequenze, ...*)

Un altro limite è dovuto al fatto che Quadretto non prevede dei sensori che consentano di interagire con l'ambiente esterno. Tutte queste possibilità saranno presentate nel capitolo *Interagire con l'ambiente*.

Nonostante le limitazioni il linguaggio si presta ad un'analisi più approfondita, affrontando alcune tematiche più avanzate, tipiche della teoria dei linguaggi di programmazione. È quanto vedremo nel prossimo capitolo *Linguaggio per un robot*.

ESERCIZI

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

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

1.2 Stabilire l'effetto dei seguenti programmi, essendo k un numero naturale.

1. k*`[FRFL]`
2. k*`[2[FR]2[FL]]`

1.3 Sia *cavallo* $\stackrel{\text{def}}{=}$ `2FRF`. Stabilire l'effetto di 2[*cavallo* `R`]

1.4 Stabilire la traiettoria generata dai seguenti programmi:

1. `2[2FR2FL]`
2. `4[2FR]`
3. `2L3[FRFL]`

1.5 Scrivere dei programmi per eseguire le seguenti transizioni di stato:

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

1.6 Quadretto si trova nello stato iniziale $(1, 1, \texttt{NORTH})$. Scrivere un programma per far percorre un percorso quadrato attorno al bordo interno del reticolo 8×8.

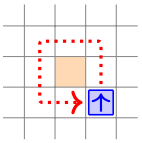
1.7 Descrivere l'effetto del seguente programma:

```
1: a ← 2FRF
2: b ← 2L
3: p ← 2[a b a]
```

1.8 Usando il programma *cavallo* definito nell'esempio 1.5.2, eseguire la transizione di stato $(1, 1, \texttt{NORTH}) \to (8, 8, \texttt{NORTH})$.

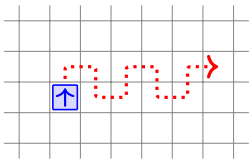
1.9 Traslare Quadretto di un passo verso destra (o sinistra), mantenendo direzione e verso di avanzamento, come indicato nella figura che segue. Traslare Quadretto di un passo in diagonale.

1.10 Muovere Quadretto lungo un percorso quadrato attorno ad una casella, come descritto dalla figura che segue.

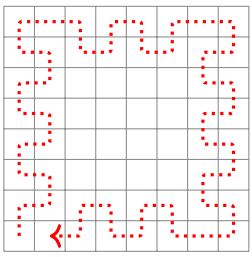


1.11 Muovere Quadretto dallo stato iniziale $(1, 1, \texttt{NORTH})$ allo stato finale $(8, 8, \texttt{NORTH})$ in modo che passi su tutte le caselle della diagonale principale $(1, 1) - (8, 8)$.

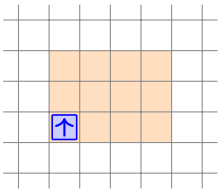
1.12 Muovere Quadretto su una traiettoria ad onda quadra, come descritto nella seguente figura:



1.13 Muovere Quadretto, a partire dallo stato iniziale $(1, 1, \texttt{NORTH})$ su una traiettoria ad onda quadra, lungo il bordo come descritto dalla figura che segue.



1.14 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.



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

- un dietrofront
- indietreggiare di un passo
- indietreggiare di n passi

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

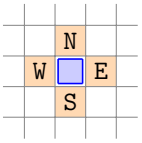
1.17 Un insieme di movimenti si dice *completo* se da ogni casella si riesce a raggiungere qualsiasi altra casella mediante una combinazione di movimenti dell'insieme. Dicesi *fortemente completo* se combinando i suoi elementi si riesce ad eseguire qualsiasi transizione di stato. Dicesi *minimale* se 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. $\{\texttt{FFL}, \texttt{FRF}\}$

1.18 Stabilire se la seguente è una metrica per lo spazio di Quadretto:

$dist(A, B)$ = numero minimo di operazioni $\texttt{F}, \texttt{B}, \texttt{L}, \texttt{R}$ per andare da A a B

1.19 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 $\mathcal{M} = \{\texttt{F}, \texttt{B}, \texttt{L}, \texttt{R}\}$, l'insieme di movimenti elementari $\mathcal{M}' = \{\texttt{N}, \texttt{E}, \texttt{S}, \texttt{W}\}$, 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).



2

LINGUAGGIO PER UN ROBOT

Il linguaggio ci permette di trattare i nostri pensieri più o meno come se fossero cose qualsiasi.

M. Minsky, *La società della mente*

L'evoluzione tecnologica dei nostri tempi, quella che identifichiamo genericamente nell'oggetto "calcolatore" e più recentemente nell'oggetto "robot". Si tratta di due diverse concretizzazioni del concetto più generale di "esecutore". Tutto ciò ha avuto un effetto collaterale: quello di 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 crea gli algoritmi per l'esecutore che risolve i problemi. La cinghia di trasmissione di questa catena di comando è costituita dal linguaggio che serve per realizzare la comunicazione fra solutore ed esecutore.

In questo capitolo viene analizzato il linguaggio `LFR` presentato nel capitolo *Muovere un robot*. Si tratta di un linguaggio molto elementare e ridotto, inadeguato per gestire molti aspetti della robotica, ma sufficientemente potente per gestire il movimento di un robot e sufficientemente interessante per affrontare alcune tematiche tipiche dei linguaggi formali.

2.1 La sintassi del linguaggio

Un metodo per precisare le frasi (corrette) di un linguaggio consiste nell'adottare un approccio generativo basato sulla definizione di un insieme di regole (dette *regole di generazione* o *regole di produzione*) mediante le quali generare le frasi del linguaggio. Queste regole costituiscono la *sintassi* (del linguaggio). Nel caso di un linguaggio di programmazione le frasi vengono dette *programmi*. Un programma può essere costruito direttamente, partendo dagli assiomi, usando le regole del linguaggio.

Come avviene per ogni linguaggio, la sintassi viene definita precisando le parole base costituenti il linguaggio (assiomi) e le regole grammaticali (regole di produzione) mediante le quali generare le frasi del linguaggio. Nel caso di Quadretto le parole chiave corrispondono ai movimenti (avanzamento e rotazioni).

Informalmente ed in modo approssimativo il linguaggio di Quadretto può essere definito informalmente come segue:

1. *Assiomi*:
   a) ogni parola chiave `F`, `B`, `L`, `R` è un programma
2. *Regole di generazione*:
   b) la concatenazione di due programmi è un programma
   c) è lecito mettere una coppia di parentesi quadre [ ] attorno ad un programma
   d) è lecito mettere un numero naturale davanti ad un programma
   e) è lecito mettere il segno `-` davanti ad un programma

Nel seguito ci riferiremo a questo linguaggio con il termine `LFR` (*L*anguage *F*or *R*obot).

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 tradizionale linguaggio della matematica, definendo il linguaggio $\mathcal{L}$ di Quadretto come segue:

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

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

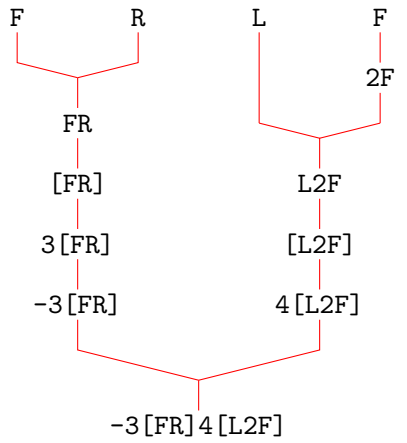
c') $\alpha \in \mathcal{L} \Rightarrow [\alpha] \in \mathcal{L}$

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

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

Seguendo queste regole è possibile descrivere graficamente la generazione di un programma mediante un *albero sintattico* come descritto nell'esempio che segue.

Esempio 2.1.1 - Il programma `-3[FR]4[L2F]` può essere generato mediante un processo di generazione descritto dal seguente albero sintattico:



□

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 aggiunto delle incoerenze (ad esempio, in questo nuovo formalismo, viene mischiato il concetto di *carattere* con il concetto di *stringa*, e mischiato il concetto di *numero* con quello di *rappresentazione* di un numero). Tutte queste ambiguità vengono risolte ricorrendo alla teoria dei "linguaggi formali", di cui daremo a seguire un breve cenno.

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 alle parole chiave. La sintassi del linguaggio `LFR` è definita come segue [1] :

Prog $\longrightarrow$ *Exp* | *Exp Prog*
Exp $\longrightarrow$ *Block* | *Num Block* | *-Block*
Block $\longrightarrow$ *Move* | `[` *Exp* `]` | *Ident*
Num $\longrightarrow$ *Cifra* | *Cifra Num*
Ident $\longrightarrow$ *Char* | *String*
String $\longrightarrow$ *Chardigit* | *Chardigit String*
Chardigit $\longrightarrow$ *Char* | *Digit*
Char $\longrightarrow$ `a` | `b` | ... | `z` | `A` | `B` | ... | `Z`
Digit $\longrightarrow$ `0` | `1` | ... | `9`
Move $\longrightarrow$ `F` | `B` | `L` | `R`

[1] Si noti che il segno - non ha il significato di operatore aritmetico di opposto di un numero ma indica l'opposto del programma che segue il segno stesso.

Esempio 2.1.2 - Basandosi sulle regole definite sopra si possono generare i seguenti programmi:

```
F
FRFL
3[12FR]2R-4[2FL]
```

□

2.2 Semantica del linguaggio

Se uno avesse iniziato a leggere dall'inizio di questo capitolo, arrivato a questo punto, sarebbe in grado di riconoscere che `2[FRFL]` è una frase (corretta) del linguaggio `LFR` ma non sarebbe in grado di interpretarla e rimarrebbe senza senso. Come ogni linguaggio, il linguaggio `LFR` di Quadretto definito nel precedente paragrafo ha bisogno di una interpretazione per risultare efficace. L'interpretazione del linguaggio avviene precisando come devono essere eseguite le frasi (programmi). Per il linguaggio `LFR` le regole sono le seguenti:

1) l'interpretazione di un singolo carattere è data dall'esecuzione della corrispondente azione di movimento
2) l'interpretazione di una sequenza di caratteri avviene interpretando in sequenza ciascun carattere
3) l'interpretazione di un numero davanti ad una parola avviene ripetendo l'esecuzione della parola tante volte pari al numero
4) l'interpretazione del segno `-` avviene eseguendo a ritroso l'inversa di ciascuna operazione del programma che segue il segno

Ad integrazione delle regole sopra elencate assumeremo la convenzione che l'esecuzione di un'azione di movimento non venga eseguita nel caso in cui Quadretto si trovi bloccato da un ostacolo.

Esempio 2.2.1 - Applicando le precedenti regole:

`F`	avanza di 1 passo
`L`	ruota a sinistra di 1 angolo retto
`FRF`	un angolo a destra
`4F`	avanza di 4 passi
`4[2FR]`	percorre un quadrato di lato uguale a 3 passi

□

L'esecuzione di un programma genera un *processo*. 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:

$$S_f = P(S_i)$$

2.3 Algebra delle rotazioni

Consideriamo l'insieme $\{\texttt{L}, \texttt{R}\}$ sul quale vogliamo definire un'operazione binaria interna, che indicheremo con il simbolo $\circ$. Se x ed y sono due elementi generici, denoteremo in notazione infissa con $x \circ y$, o semplicemente con xy, l'operazione fra x ed y. Vogliamo che la logica che regge questa operazione, ossia il criterio in base al quale calcolare il risultato, si fondi sull'equivalenza fra il risultato e lo stato finale che si ottiene eseguendo il programma; in altri termini il risultato di $x \circ y$ dovrà corrispondere a "esegui la rotazione x e poi la rotazione y". Si vede subito che `L`$\circ$`R`, `R`$\circ$`L`, `L`$\circ$`L`, `R`$\circ$`R` 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 base a rotazioni di Quadretto. Disponiamo così di uno strumento che ci permette di valutare qualsiasi espressione, 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` (moltiplicazione, raggruppamento, inversione), ottenendo così espressioni anche molto articolate (ad esempio `3RL4[RI-[2L]]`).

Esempio 2.3.1 - Il processo di valutazione dell'espressione RRLOIRLLL può essere svolto a coppie di operandi, in parallelo; indicando con il simbolo → il passaggio al risultato che si ottiene dal processo di valutazione di un'espressione, 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 2.3.2 - Basandosi sull'operazione di moltiplicazione di un numero naturale per un programma si può interpretare all'interno dell'algebra dell'operazione ∘ anche espressioni della forma 5R, come descritto nel seguente calcolo:

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

□

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 ∘) si possono evitare alcuni calcoli, eseguendo delle trasformazioni 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. se x è un movimento elementare, [x] può essere sostituito con x

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

Esempio 2.3.3 - Applicando le precedenti regole di trasformazione si possono svolgere i seguenti calcoli:

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

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

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

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

□

Esempio 2.3.4 - 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 2.3.1 può essere semplificata e valutata come segue:

$$\texttt{RRLOIRLLL} \rightarrow \texttt{R}\underline{\texttt{RLO}}\texttt{IR}\underline{\texttt{LLL}} \rightarrow \underline{\texttt{RRRR}}\texttt{R} \rightarrow \texttt{R}$$

□

2.4 Algebra dei programmi

Le operazioni di concatenazione, di ripetizione, di raggruppamento e di inversione sopra descritte costituiscono una particolare algebra con regole di trasformazione e semplificazione che per programmi costanti possono essere riassunte come segue:

Regole di trasformazione: se m, n sono due numeri naturali ed α un generico programma

1. è lecito sostituire n caratteri x consecutivi con nx
2. è lecito sostituire n parole α consecutive con $n[\alpha]$
3. è lecito sostituire $n\alpha\, m\alpha$ con $(m+n)\alpha$

Con queste regole si possono costruire programmi di movimento che sono del tutto equivalenti ad una struttura dati composta da atomi costituiti dai movimenti elementari.

Esempio 2.4.1 - Con riferimento a Quadretto, a seguire sono riportate, in corrispondenza di ciascune delle regole di trasformazione riportate sopra, una loro applicazione:

```
      FFRFFF  →  2FR3F
   FLFFFRFFFR  →  FL2[FFFR]
2[FLF]3[FLF]  →  5[FLF]
```

□

L'operazione *inversa* gode di alcune proprietà, di facile dimostrazione.

PROPRIETÀ 1. Data una generica azione x, valgono le seguenti proprietà:

$$x[-x] = \theta$$

$$-[-x] = x$$

PROPRIETÀ 2. Date le azioni $x_1, x_2, \ldots, x_n$, vale la seguente proprietà:

$$-[x_1\, x_2\, \ldots\, x_n] = [-x_n\, -x_{n-1}\, \ldots\, -x_2\, -x_1]$$

PROPRIETÀ 3. Data una generica azione x ed un numero naturale n, vale la seguente proprietà:

$$-[n\, x] = n\, [-x]$$

Esempio 2.4.2 - Applicando le precedenti proprietà si ha:

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

□

2.5 Analisi dei programmi

Analizzare un programma significa valutare sulla carta, senza eseguire il programma, le principali caratteristiche che emergono dalla sua esecuzione. Questa forma di analisi implica capacità di astrazione, 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 programma sono riportate a seguire, dove ammetteremo l'ipotesi che le caselle della scacchiera sulla quale si muove Quadretto siano libere da altri oggetti.

Invarianza di un programma

Un programma è *invariante rispetto allo stato* se non modifica lo stato, ossia se lo stato finale raggiunto dopo aver eseguito il programma risulta uguale a quello iniziale.

Esempio 2.5.1 - Il programma `4[2FR]` è invariante rispetto allo stato. □

Equivalenza di programmi

Una particolare forma di analisi dei programmi riguarda la relazione di *equivalenza* fra coppie di programmi. In particolare, dati due programmi P_1 e P_2 ci proponiamo di dare significato ad una scrittura come $P_1 = P_2$. È evidente che `2FLRFR` = `2FLRFR`, ma `FFLRFR` = `3FR` ? La questione trova risposta definendo il concetto di equivalenza, come segue. In modo approssimativo due programmi si dicono *equivalenti* se hanno effetti similari. Per i programmi per Quadretto scritti in linguaggio `LFR` si possono distinguere diverse forme di equivalenza:

1. equivalenza di *scrittura*: quando i programmi sono descritti da stringhe identiche; ad esempio: `2FLRFR` e `2FLRFR`
2. equivalenza di *azioni*: quando l'esecuzione dei programmi fa compiere a Quadretto le stesse azioni; ad esempio: `2[FFR]` e `FFRFFR`
3. equivalenza di *percorso*: quando l'esecuzione dei programmi fa compiere a Quadretto lo stesso percorso, portandolo nello stesso stato finale; ad esempio: `FF2RF` e `2F2LF`
4. equivalenza di *stato finale*: quando l'esecuzione dei programmi porta Quadretto allo stesso stato finale; ad esempio: `FFRF` e `LFRFFR`

È facile dimostrare che le quattro forme di relazione 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, la seconda implica la terza e la terza implica la quarta.

Corrispondentemente alle quattro forme di equivalenza sopra definite si usano anche le seguenti definizioni alternative. Due programmi si dicono *uguali* se sono definiti da stringhe uguali; si dicono *equivalenti* se generano la stessa sequenza di azioni; si dicono *funzionalmente equivalenti* se, a partire da stati iniziali uguali, fanno compiere lo stesso percorso e portano a stati finali uguali; si dicono *equivalenti rispetto allo stato* se, a partire da stati iniziali

uguali, portano a stati finali uguali. L'uguaglianza è la forma più restrittiva di equivalenza; due programmi equivalenti sono anche funzionalmente equivalenti; l'equivalenza rispetto allo stato è la forma più debole di equivalenza ed è implicata da tutte le altre forme.

Esempio 2.5.2 - I due programmi P_1 e P_2 che seguono sono equivalenti ma non uguali:

$$P_1 \stackrel{\text{def}}{=} \texttt{3F2L}$$
$$P_2 \stackrel{\text{def}}{=} \texttt{FFFLL}$$

I due programmi Q_1 e Q_2 che seguono sono funzionalmente equivalenti ma non equivalenti:

$$Q_1 \stackrel{\text{def}}{=} \texttt{FR2F}$$
$$Q_2 \stackrel{\text{def}}{=} \texttt{2[FR]LF}$$

□

Complessità di un programma

La *complessità* di un programma è data dal numero di operazioni elementari che vengono eseguite nell'esecuzione del programma. Nell'ipotesi che le azioni elementari impieghino lo stesso tempo di esecuzione, la complessità risulta proporzionale al tempo di esecuzione del programma. Dati due programmi equivalenti, per questioni di efficienza, risulta preferibile quello avente la minore complessità.

Esempio 2.5.3 - Il programma `4[2FR]` ha complessità 12 corrispondente all'espressione $4(2+1)$, pari alla lunghezza della stringa `FFRFFRFFRFFR` ottenuta espandendo il programma in una equivalente stringa piatta. □

2.6 Trasformazioni dei programmi

Consideriamo due stringhe (programmi) α e β ed una generica trasformazione $\alpha \to \beta$ che trasforma la stringa α in β. Ci proponiamo di individuare delle regole che guidino queste trasformazioni, volendo che queste trasformazioni risultino utili ed interessanti. Queste trasformazioni possono fondarsi su un dato criterio di equivalenza; il più interessante, ai fini dell'analisi dei programmi per Quadretto, risulta il criterio di equivalenza di percorso, che assumiamo nelle prossime considerazioni; in altri termini vuol dire che la trasformazione $\alpha \to \beta$ è ammessa se l'esecuzione di α e di β fa compiere a Quadretto lo stesso percorso; inoltre, per un principio di utilità, cercheremo delle regole che *accorciano*, ossia la stringa β dovrà risultare di lunghezza minore o uguale a quella di β.

Semplificazione di un programma

Semplificare un programma significa trasformarlo in uno equivalente di complessità minore. La semplificazione si basa sulle seguenti regole di cancellazione e trasformazione:

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

dove con ϵ si denota la *stringa vuota* costituita da nessun carattere. Queste regole di semplificazione possono essere utilizzate unitamente alle regole di trasformazione descritte nel paragrafo 2.4, generando così dei processi di trasformazione che espandono o comprimono una stringa in una equivalente.

Esempio 2.6.1 - A seguire sono descritte due catene di trasformazioni (una di compressione e l'altra di espansione):

$$\begin{aligned}
&\texttt{F2FR3F2R} \rightarrow \texttt{3FR3FRR} \rightarrow \texttt{2[3FR]R} \\
&\texttt{2[L3FR]} \rightarrow \texttt{2[LFFFR]} \rightarrow \texttt{LFFFRLFFFR}
\end{aligned}$$

□

Problema 2.6.1 Semplificare il programma `2[FR]2LF`.

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

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

□

Problema 2.6.2 Semplificare il programma `FFR-[BR]`.

Soluzione. : Semplificando:

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

□

Problema 2.6.3 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 dai seguenti due programmi P_1 e P_2 funzionalmente equivalenti:

$$P_1 \stackrel{\text{def}}{=} \texttt{2R3F2L}$$
$$P_2 \stackrel{\text{def}}{=} \texttt{3[2RF2R]}$$

Questi due programmi hanno le seguenti complessità:

$$compl(P_1) = 2 + 3 + 2 = 8$$
$$compl(P_2) = 3(2 + 1 + 2) = 15$$

Da ciò si conclude che P_1 è più efficiente rispetto a P_2. Il programma P_2 può essere semplificato ed ottimizzato come segue (sono state sottolineate le parti di programma 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 P_1. □

2.7 Compilazione ed ottimizzazione dei programmi

Abbiamo visto, all'inizio del capitolo, che Quadretto è in grado di eseguire in sequenza le azioni elementari `F`, `B`, `L`, `R`; quindi esso è in grado di eseguire *nativamente*, ossia senza alcun intervento di intermediazione esterna, delle stringhe della forma `FFRFFLF`. Programmi come `2[3FR]` non sono invece compresi da Quadretto ma richiedono una fase di preelaborazione che fornisca una equivalente stringa piatta `FFFRFFFR`, direttamente interpretabile ed eseguibile da Quadretto. A seguire la fase di compilazione viene solitamente svolta la fase di *ottimizzazione* che consiste nel semplificare il codice 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. Ad esempio, la sottostringa `LR` può essere eliminata, la sottostringa `RRRRR` può essere semplificata in `R` e la sottostringa `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 [2] ed è descritta nella figura 2.1.

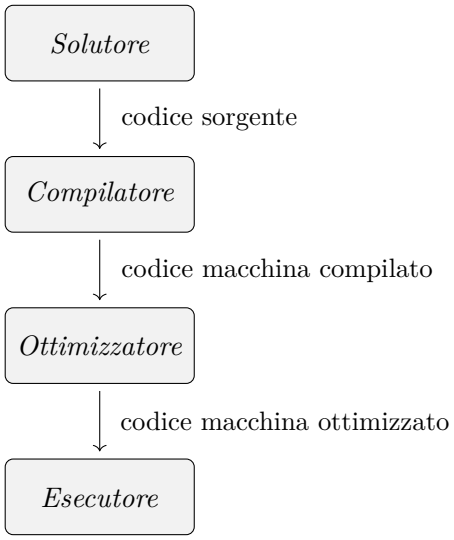


Figura 2.1: Schema di compilazione di un programma: dal codice sorgente creato dal solutore al codice macchina ottimizzato eseguito dall'esecutore.

Esempio 2.7.1 - 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 dipende dagli oggetti presenti nelle caselle della scacchiera.

Osservazione. La fase di compilazione genera un programma equivalente al sorgente, mentre la fase di ottimizzazione produce un programma ottimizzato funzionalmente equivalente al compilato.

[2] Nel caso di Quadretto il linguaggio macchina è un sottoinsieme del linguaggio di programmazione `LFR`; per la maggior parte degli elaboratori il linguaggio macchina è distinto dal linguaggio di programmazione utilizzato dal solutore.

2.8 I processi

L'esecuzione di un programma P fa compiere a Quadretto un percorso λ che dipende, oltre che da P, dallo stato iniziale s_0 di Quadretto. La forma del percorso λ è individuata dal programma P, mentre la localizzazione (posizione ed orientamento) di λ 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 λ si utilizza un *grafo di transizione di stato* come quello descritto nella figura 2.2.

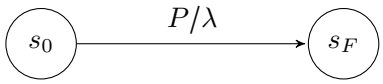


Figura 2.2: Grafo di transizione dallo stato s_0 allo stato s_F eseguendo il programma P.

Un grafo di transizione di stato come quello riportato nella figura 2.2 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 λ 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, λ, 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, λ 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 λ 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 λ.

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).

ESERCIZI

2.1 Stabilire se le seguenti stringhe sono dei programmi corretti secondo la sintassi del linguaggio `LFR`. In caso affermativo generarli a partire dall'assioma *Program*, secondo le regole di derivazione definite nel par. 2.1 e precisarne la semantica, compilandoli in una sequenza di azioni elementari:

1. `-2FR`
2. `-2[FR]`
3. `2-[FR]`
4. `2-FR`
5. `-[2FR]`
6. `-[2[FR]]`
7. `2[-FR]`
8. `2[-[FR]]`

2.2 Studiare le proprietà (commutatività ed associatività) dell'operazione $\circ$ sull'insieme $\mathcal{R}$ delle rotazioni.

2.3 Calcolare le seguenti espressioni di rotazioni:

1. `7RLI`
2. `3R2LIR`
3. `2[3RL]3[2IL]`
4. `4[LIR]7RL`
5. `3[R2[RIL]]`

2.4 Sia M un movimento di $\mathcal{M} = \{\mathtt{F}, \mathtt{B}, \mathtt{L}, \mathtt{R}\}$ 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. $m[nM] \equiv (mn)M$

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

2.6 Descrivere mediante degli alberi sintattici la generazione dei seguenti programmi:

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

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

1. `3FRF2L`

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

2.8 Determinare il programma di minima complessità che muove Quadretto dallo stato iniziale $(1, 1, N)$ allo stato $(8, 8, N)$

2.9 Compilare ed ottimizzare i seguenti programmi:

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

2.10 Stabilire se il programma `FR` è equivalente al programma `RF`.

2.11 Determinare un insieme minimale $\mathcal{M}_0$ di movimenti potenzialmente equivalente all'insieme dei movimenti $\mathcal{M} = \{\mathtt{F}, \mathtt{B}, \mathtt{L}, \mathtt{R}\}$. Esprimere ciascun movimento di $\mathcal{M} \setminus \mathcal{M}_0$ mediante combinazione dei movimenti di $\mathcal{M}_0$.

3

INTERAGIRE CON L'AMBIENTE

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*

Il movimento diventa interessante se l'entità che si muove riesce ad interagire con l'ambiente circostante o 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 permette un cambio di marcia nella programmazione del robot.

In questo capitolo faremo ancora riferimento a Quadretto. Per programmare i robot dotati di sensori ed attuatori serve un linguaggio più potente di quello visto al capitolo precedente per muovere Quadretto; inoltre, anche gli algoritmi richiesti sono più complessi e richiedono il repertorio dei controlli condizionali e ciclici. In questo modo gli algoritmi potranno essere facilmente codificati in un linguaggio di programmazione quali, ad esempio, Python o C.

3.1 Robot

Una definizione del termine robot che coglie efficacemente il significato che noi oggi assegnamo a questo termine, in particolare nel contesto della robotica educativa, la possiamo dedurre da quanto affermava George Bekey agli inizi di questo secolo: definiva un robot come *una macchina che sente, pensa ed agisce.* Questa definizione ha il pregio di individuare le tre principali componenti di un robot:

- *sensori*: sono le componenti corrispondenti ai sensi dell'uomo (tatto, vista, udito, olfatto e gusto); permettono di acquisire informazioni dall'ambiente circostante (sensori di contatto, di temperatura, di colore, di vista, di suono, ...)
- *attuatori*: sono le componenti che permettono di agire sull'ambiente esterno, azionando motori, ruote, pinze, ...
- *sistema di controllo*: espleta le funzioni che in un essere umano sono svolte dal cervello; permette di eseguire delle elaborazioni e di prendere delle decisioni su come/quando azionare gli attuatori

Le caratteristiche sopra descritte sono sintetizzate nello schema riportato nella figura 3.1.

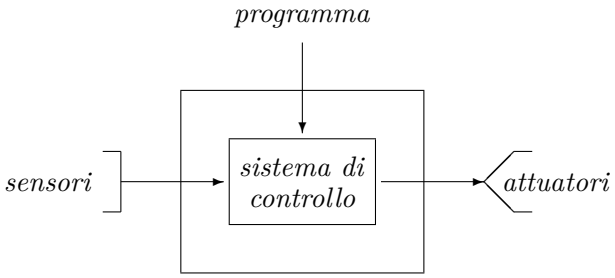


Figura 3.1: Struttura generale di un robot.

Tipologie dei robot

Nella letteratura tecnica e scientifica tradizionale i robot vengono classificati in base a criteri costruttivi, come segue:

- robot *fisico*: oggetto reale, costituito da componenti meccaniche ed elettroniche, che si muove nello spazio fisico (su un tavolo, in una stanza, sul terreno, nello spazio)
- robot *virtuale*: oggetto grafico che si muove sul video di un computer, con il quale si interagisce mediante l'uso di periferiche (tastiera, mouse, joystick, ...)

Questa dicotomia fisico-virtuale sta diventando sempre più labile; dal punto di vista didattico e metodologico e dal punto di vista della sua programmazione risulta più utile la seguente classificazione basata sulle capacità dei robot:

- robot *movente*: entità in grado di muoversi in un ambiente mediante l'uso di *motori*
- robot *sensibile*: costituisce un'evoluzione del robot meccanico in quanto è dotato di *sensori* che gli permettono di percepire alcuni parametri (temperatura, luce, pressione, ...) dell'ambiente esterno
- robot *agente*: entità in grado di esplicare azioni e modificare l'ambiente esterno mediante l'uso di *attuatori*
- robot *intelligente*: entità che reagisce a delle situazioni esterne, in modo tale da *sembrare* [1] un'entità intelligente e dotata di volontà ed autonomia operativa.

Motori, sensori ed attuatori di cui si parla sopra possono essere sia fisici (per i robot fisici) che virtuali (per i robot virtuali). Le potenzialità del robot dipendono principalmente dalle capacità elaborative interne, dalla tipologia e dalla sensibilità, precisione e potenza di questi componenti.

Osservazione. 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 indicata la posticondizione che risulterà verificata alla fine dell'esecuzione dell'algoritmo, specificando la clausola *Ensure* che esprime una postcondizione che risulterà sicuramente verificata al termine dell'esecuzione dell'algoritmo.

3.2 La programmazione dei movimenti

Un robot va opportunamente programmato in modo da definirne il comportamento, a seconda di come deve muoversi nel contesto esterno che incontrerà. Per movimentare Quadretto sono predisposti i seguenti comandi [2] :

- *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

[1] Dietro alla parola *sembrare* si nascondono delicate questioni e difficili problemi di ordine tecnico, sociale, culturale, filosofico ed etico. Qui il discorso si fa estremamente impegnativo ed arduo in quanto si entra nelle tematiche e problematiche relative alla cosidetta *intelligenza artificiale*.

[2] 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*().

Esempio 3.2.1 - Usando i precedenti comandi si può far fare a Quadretto un percorso a mossa di cavallo degli scacchi mediante la seguente porzione di algoritmo:

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

Questo programma corrisponde a `2FRF` del linguaggio `LFR`. □

Per aumentare la leggibilità, spesso si denomina una porzione di algoritmo mediante un identificatore significativo.

Esempio 3.2.2 - La porzione di algoritmo vista nell'esempio 3.2.1 può essere denominata *cavallo*:

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

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

Con il comando *cavallo* di ottiene l'esecuzione delle azioni descritte nell'algoritmo. □

Esempio 3.2.3 - Il seguente semplice algoritmo inverte il senso di avanzamento di Quadretto.

Algoritmo 2 - inverte il senso di avanzamento - *inverti*

```
1: left
2: left
```

□

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

Esempio 3.2.4 - Per percorrere un quadrato di lato 3 caselle si può ricorrere al seguente algoritmo che richiama l'algoritmo *cavallo*.

Algoritmo 3 - 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 3.2.5 - Il seguente algoritmo permette di avanzare Quadretto di un numero n di passi, specificato al momento della chiamata dell'algoritmo.

Algoritmo 4 - 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 *avanti*(5), con l'effetto di far avanzare Quadretto di 5 passi nella direzione corrente. □

Problema 3.2.1 Percorrere il bordo di un quadrato di lato n.

Soluzione. Il problema è risolto dal seguente algoritmo:

Algoritmo 5 - 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
```

□

3.3 Muoversi conoscendo lo stato

In ogni istante la situazione di Quadretto è completamente descritta dal suo *stato*, costituito dalla *posizione* in cui si trova e dal *verso* (`NORTH`, `WEST`, `SOUTH`, `EAST`) 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 $(3, 4, \texttt{NORTH})$ denota lo stato di Quadretto quando è posto nella casella di coordinate $(3, 4)$ e rivolto verso nord. 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.

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:

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

Per Quadretto le coordinate della posizione attuale ed il verso attuale di avanzamento sono forniti dalle seguenti funzioni senza argomenti:

- *posx* : coordinata di riga
- *posy* : coordinata di colonna
- *versus* : verso di avanzamento

Problema 3.3.1 Stabilire se Quadretto è posizionato sul bordo della scacchiera.

Soluzione. Ricorrendo ai sensori di stato *posx* e *posy*, la condizione di essere sul bordo è espressa dalla seguente espressione:

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

□

Problema 3.3.2 Orientare Quadretto in un dato verso v.

Soluzione. Il problema è risolto dall'algoritmo 6.

Algoritmo 6 - orienta verso v - $orienta(v)$

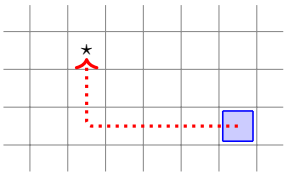
Input: verso v a cui orientarsi
Ensure: Quadretto è orientato nel verso v

```
1: while versus ≠ v do
2:     left
3: end while
```

□

Problema 3.3.3 Raggiungere una data 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 7.



Algoritmo 7 - 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*($posx < x$, `EAST`, `WEST`))
2: *avanti*($|posx - x|$)
3: *orienta*(*if*($posy < y$, `NORTH`, `SOUTH`))
4: *avanti*($|posy - y|$)

□

Problema 3.3.4 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 8 - 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 Acquisire informazioni dall'ambiente

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

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 esso. Un sensore è caratterizzato dalla diversa tipologia di informazione esterna che è in grado di recepire, inducendo una classificazione dei sensori stessi. Nei robot più elementari (come ad esempio Quadretto) si individuano le seguenti diverse tipologie di sensori:

1. sensore di *contatto*: sente quando il robot ha una sua parte (solitamente la parte frontale) in contatto con un oggetto
2. sensore di *tatto*: riesce a qualificare la diversa tipologia dell'oggetto con il quale è in contatto
3. sensore di *pressione*: sente se davanti c'è un oggetto che non può essere spinto
4. sensore di *distanza*: fornisce la distanza dall'oggetto posto davanti al robot nella direzione di avanzamento
5. sensore di *colore*: percepisce il colore dell'oggetto davanti

Ciascuna di queste tipologie di sensori di Quadretto è descritta a seguire.

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 9 fa compiere a Quadretto 1 passo, facendo un dietro-front nel caso si trovi inizialmente a contatto con l'ostacolo.

Algoritmo 9 - avanza di un passo - *passo*

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

□

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 10 avanza Quadretto nel caso in cui non sia a contatto frontale con il muro.

Algoritmo 10 - 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 11 avanza Quadretto, spingendo in avanti eventuali blocchi, fino ad arrivare ad una situazione di impossibilità ad avanzare.

Algoritmo 11 - 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 12 avanza Quadretto fino ad arrivare ad una distanza d dall'oggetto davanti.

Algoritmo 12 - 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

Esempio 3.4.5 - L'algoritmo 13 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 13 - orienta verso il colore c - *orientaColore*(c)

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

□

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 $(1, 1, \texttt{NORTH})$.

Soluzione. Il problema proposto può essere risolto risolto combinando l'algoritmo *raggiungi*(x, y) (Algoritmo 8) e l'algoritmo *orienta*(v) (Algoritmo 6). Una soluzione alternativa può essere sviluppata direttamente utilizzando i sensori di stato e di contatto come descritto nell'algoritmo 14.

Algoritmo 14 - porta nello stato $(1, 1, \texttt{NORTH})$ - *home*

Require: Quadretto si trova all'interno del reticolo
Ensure: Quadretto si trova nello stato $(1, 1, \texttt{NORTH})$
1: *orienta*(SOUTH)
2: **for** 2 **times**
3: **while** $\neg$ *touch* **do**
4: *forward*
5: **end while**
6: *right*
7: **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 15 realizza un sensore virtuale *toccaSinistra* che rileva se il robot è a contatto con un ostacolo sul suo fianco sinistro.

Algoritmo 15 - *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 16). Si ha così a disposizione un nuovo *sensore virtuale tocca*(l) che permette di stabilire se Quadretto è in contatto sul lato l.

Algoritmo 16 - test di contatto sul lato l - *tocca*(l)

```
Input: lato l sul quale sentire il contatto
 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
```

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 la situazione iniziale. La soluzione è descritta nell'algoritmo 17.

Algoritmo 17 - *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

□

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).

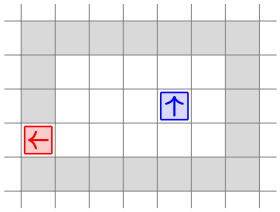


Figura 3.2: Il problema della chiusura della stanza.

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

Algoritmo 18 - *chiudi stanza*

Input: 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 18 è concentrato nell'azione *muoviti*, dettagliata nell'algoritmo 19.

Algoritmo 19 - *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 ostacoli

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'ostacolo

Usando il sensore di contatto mediante il controllo *tocca* è possibile far avanzare Quadretto fino ad entrare in contatto con un ostacolo che si trova davanti lungo la direzione di avanzamento. Il procedimento descritto presuppone che l'ostacolo 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'ostacolo, non viene svolta alcuna azione. Il procedimento è descritto nell'algoritmo 20.

Algoritmo 20 - porta a contatto con il muro - *contatta*

Ensure: *touch*
1: **while** $\neg$ *touch* **do**
2: *forward*
3: **end while**

Liberarsi da un ostacolo

Se Quadretto si trova a contatto frontale con l'ostacolo, per intraprendere un'azione di avanzamento è necessario ruotare fino a trovare una direzione di avanzamento libera. Il procedimento è descritto nell'algoritmo 21.

Algoritmo 21 - libera il lato frontale dal contatto col muro - *libera*

Input: c'è (almeno) un lato libero da ostacoli
Ensure: $\neg$ *touch*
1: **while** *touch* **do**
2: *right*
3: **end while**

Strisciare lungo un ostacolo

In molte situazioni si presenta l'esigenza di avanzare mantenendosi a contatto su un fianco con un ostacolo. Consideriamo la precondizione di essere a contatto sul fianco sinistro, di avere libero il lato frontale e di voler avanzare di 1 passo in modo che il fianco sinistro di Quadretto si sposti a contatto del successivo tratto di muro. Per poter eseguire in modo ciclico questa azione è necessario, dopo ogni passo, riconquistare la precondizione *tocca*(`LEFT`) $\wedge$ $\neg$ *tocca*(`FRONT`). 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 le seguenti tre:

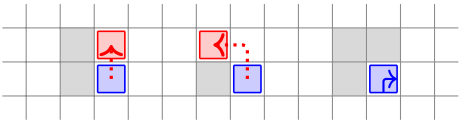


Figura 3.3: Quadretto avanza di 1 passo affiancato ad un ostacolo.

Il procedimento è descritto nell'algoritmo 22.

Algoritmo 22 - *striscia*

Input: *tocca*(LEFT)
Ensure: *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 ostacoli

Combinando le azioni *contatta*, *libera* e *striscia* esaminate nel precedente paragrafo si riesce a risolvere interessanti problemi di movimento fra gli ostacoli, come descritto nei seguenti sottoparagrafi.

Accostare un ostacolo

In molte situazioni di movimento, per poter evitare, superare o circumnavigare un ostacolo si rende necessaria una azione preparatoria che consiste nell'avvicinarsi all'ostacolo mettendosi in contatto frontale con esso. Il procedimento di accostamento può essere scomposto come segue:

```
1: ruota per cercare un eventuale contatto locale
2: if non hai trovato il contatto then
3:     avanza fino ad entrare in contatto con l'ostacolo davanti
4: end if
```

Il procedimento è dettagliato nell'algoritmo 23.

Algoritmo 23 - *accosta*

Ensure: *touch*

```
1: k ← 0
2: while ¬ touch ∧ (k < 4) do
3:     right
4:     k ← k + 1
5: end while
6: if ¬ touch then
7:     contatta
8: end if
```

Affiancare un ostacolo

Questa azione è descritta nella figura che segue: quadretto si accosta all'ostacolo e poi ruota per liberare la testa dalla condizione di contatto frontale, in modo da poter avanzare.

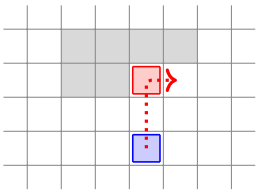


Figura 3.4: Quadretto accosta ed affianca l'ostacolo posto davanti lungo la direzione di avanzamento.

Il procedimento di affiancamento può essere scomposto nelle seguenti due azioni:

1: entra in contatto frontale con l'ostacolo
2: ruota a destra fino a liberarti dal contatto frontale

Si riconosce facilmente che queste due azioni corrispondono proprio alle due azioni *accosta* e *libera* precedentemente esaminate; pertanto il procedimento complessivo di affiancamento può essere descritto come riportato nell'algoritmo 24.

Algoritmo 24 - *affianca*

Ensure: $tocca(\texttt{LEFT}) \wedge \neg\, tocca(\texttt{FRONT})$
1: *accosta*
2: *libera*

Costeggiare un ostacolo

Supponiamo che Quadretto si trovi a contatto o davanti ad un ostacolo, lungo la direzione di avanzamento iniziale ed inoltre che l'ostacolo sia isolato dal muro perimetrale. Per costeggiare l'ostacolo è necessario ruotare fino a trovare una direzione di avanzamento libera, rimanendo a contatto sul fianco sinistro. Una tale situazione è descritta nella figura 3.5 dove la linea descrive il tragitto di costeggiamento della posizione del baricentro di Quadretto.

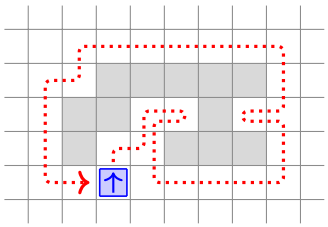


Figura 3.5: Percorso di costeggiamento di un ostacolo.

L'idea fondamentale sulla quale fondare l'algoritmo per costeggiare l'ostacolo è la seguente: *affianca l'ostacolo ed avanza mantenendoti a contatto con il fianco sinistro.* Lo schema generale del procedimento è descritto nell'algoritmo 25. Il ciclo infinito comporta che Quadretto costeggi l'ostacolo indefinitamente.

Algoritmo 25 - *costeggia*

Input: Q si trova a contatto o davanti ad un ostacolo
Ensure: Q circumnaviga l'ostacolo indefinitamente
1: *affianca*
2: **loop**
3: *striscia*
4: **end loop**

Circumnavigare un ostacolo

Una variazione del problema di costeggiamento consiste nel circumnavigare l'ostacolo, fermandosi, dopo un giro, alla posizione del primo contatto. A questo fine è sufficiente memorizzare la posizione di partenza e costeggiare l'ostacolo fino a raggiungere la posizione iniziale. Il procedimento è descritto nell'algoritmo 26.

Algoritmo 26 - *circumnaviga*

Input: Q si trova a contatto o davanti ad un ostacolo
Ensure: Q costeggia l'ostacolo fino a tornare al punto di primo contatto
1: *affianca*
2: $(x_i, y_i) \leftarrow (posx, posy)$ ▷ posizione iniziale
3: **repeat**
4: *striscia*
5: **until** $(posx, posy) = (x_i, y_i)$

Superare un ostacolo

Consideriamo il problema di superare un ostacolo posto davanti a Quadretto ed isolato dal bordo della scacchiera, come descritto nella figura 3.6.

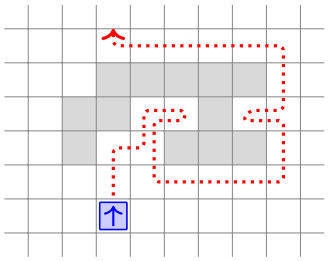


Figura 3.6: Percorso di Quadretto per superare un ostacolo.

Una volta costeggiata una parte dell'ostacolo, trovandosi dall'altra parte dell'ostacolo, 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'ostacolo può essere scomposto in sotto azioni, come descritto nell'algoritmo 27.

Algoritmo 27 - *supera ostacolo* - ver.1

1: avanza fino ad arrivare in contatto con l'ostacolo 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 più raffinato l'algoritmo si esprime come descritto nel seguente algoritmo 28.

Algoritmo 28 - *supera ostacolo* - ver.2

Input: Q si trova davanti ad un ostacolo ed isolato dal bordo
Ensure: Q supera il muro davanti e si rimette sulla stessa linea di avanzamento

1: *contatta*
2: memorizza lo stato iniziale (x_0, y_0, v_0)
3: *superato* ← `FALSE`
4: **while** ¬ *superato* **do**
5: *striscia*
6: **if** $\mathcal{C}$ **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:

$$((v_0 = \texttt{NORTH}) \wedge (x_c = x_0) \wedge (yc > y0)) \vee$$
$$((v_0 = \texttt{SOUTH}) \wedge (x_c = x_0) \wedge (yc < y0)) \vee$$
$$((v_0 = \texttt{EAST}) \wedge (x_c > x_0) \wedge (yc = y0)) \vee$$
$$((v_0 = \texttt{WEST}) \wedge (x_c < x_0) \wedge (yc = y0))$$

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 29.

Algoritmo 29 - *raggiungi obiettivo*

```
1: while ¬ raggiunto obiettivo do
2:     muoviti per raggiungerlo
3: end while
```

Consideriamo il problema in cui Quadretto deve esplorare l'ambiente circostante e ricercare un oggetto, all'interno della scacchiera, portandosi a contatto con esso. 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.

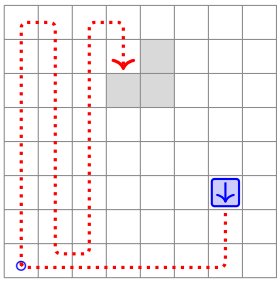


Figura 3.7: Percorso a serpentina alla ricerca di un oggetto.

La struttura generale dell'algoritmo basato su questa strategia può essere espressa come indicato nell'algoritmo 30.

Algoritmo 30 - *ricerca oggetto*

Input: 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 31 e 32, nei quali emergono i tre sottoproblemi P_4, P_5 e P_6.

Algoritmo 31 - *portati nell'angolo avanti a destra*

1: **for** 2 **times**
2: avanza fino al bordo $[P_4]$
3: *right*
4: **end for**

Algoritmo 32 - *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):

$$\begin{aligned} \mathit{toccaBordo} &\stackrel{\text{def}}{=} ((x = 1) \wedge (v = \texttt{WEST})) \vee ((x = 8) \wedge (v = \texttt{EAST})) \vee \\ &\quad ((y = 1) \wedge (v = \texttt{SOUTH})) \vee ((y = 8) \wedge (v = \texttt{NORTH})) \\ \mathit{toccaOstacolo} &\stackrel{\text{def}}{=} \mathit{touch} \wedge \neg \mathit{toccaBordo} \end{aligned}$$

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 33.

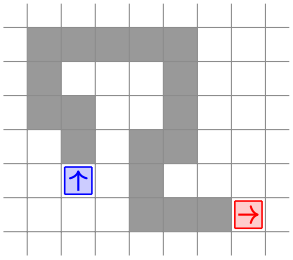
Algoritmo 33 - *ricerca oggetto*

Input: c'è un oggetto nello spazio raggiungibile da Quadretto
Ensure: Quadretto è a contatto frontale con l'oggetto

```
1: for 2 times
2:    while ¬ toccaOstacolo ∧ ¬ toccaBordo do
3:       forward
4:    end while
5:    if ¬ toccaOstacolo then
6:       right
7:    end if
8: end for
9: curva ← TRUE
10: while ¬ toccaOstacolo do
11:    forward
12:    if toccaBordo then
13:       for 2 times
14:          if ¬ toccaOstacolo then
15:             if curva then
16:                right
17:             else
18:                left
19:             end if
20:          end if
21:          if ¬ toccaOstacolo then
22:             forward
23:          end if
24:       end for
25:       curva ← ¬ curva
26:    end if
27: end while
```

3.9 Seguire una linea

Un classico problema per robot che si muovono su un piano consiste nel far percorrere al robot 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 *color* 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.



La strategia per percorrere la linea è descritta nell'algoritmo 34.

Algoritmo 34 - *segui linea*

Input: Quadretto è all'inizio della linea
Ensure: Quadretto è uscito alla fine della linea

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

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

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 35).

Algoritmo 35 - metodo di Tremaux

1: Quando ti trovi a 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.

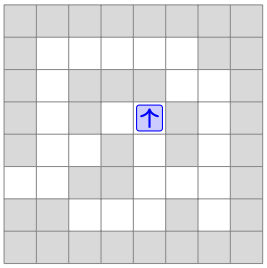
Il precedente algoritmo può essere formulato anche nella seguente equivalente versione, nota come *regola della mano destra* (algoritmo 36).

Algoritmo 36 - metodo della mano destra

1: Avanza toccando con la mano destra il muro sul fianco destro.

Nei due algoritmi 35 e 36 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 36 al caso in cui, 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 37 - *esci dal labirinto*

Input: (Quadretto tocca il labirinto) $\wedge$ (il labirinto è connesso)
Ensure: Quadretto è a contatto frontale con l'oggetto
1: **while** $\neg$ (*sul bordo*) **do**
2: *striscia*
3: **end while**

Il predicato *sul bordo* può essere definito come segue, dove con (x, y) si denotano le coordinate correnti di Quadretto:

$$\textit{sul bordo} \stackrel{\text{def}}{=} (x = 1) \vee (x = 8) \vee (y = 1) \vee (y = 8)$$

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.

Osservazione. Nei robot reali si devono gestire ulteriori caratteristiche dei sensori:

- numero dei sensori (1, 2, array)
- distanza fra i sensori (se più di 1)
- posizionamento dei sensori (davanti, laterali, ...)
- sensibilità dei sensori di colore (bianco/nero, scala di grigi, colori)
- cono di rilevamento dei sensori di distanza (angolo ed apotema)

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 percoso 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 muro

3.6 Muovere Quadretto indefinitamente avanti-indietro fra due ostacoli 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 il muro (interno o esterno).

3.8 Muovere Quadretto a contatto frontale con il muro più vicino (interno o esterno).

3.9 Determinare la minima distanza di Quadretto dai bordi (0 se è sul bordo).

3.10 Stabilire se sulla scacchiera sono presenti oggetti.

3.11 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).

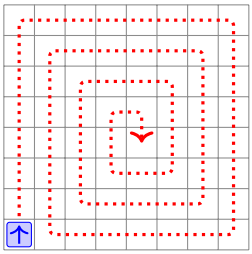
3.14 Traslare Quadretto di n passi in un dato verso v.

3.15 Traslare Quadretto di n passi a sinistra (a destra).

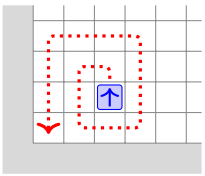
3.16 Avanzare Quadretto di n passi, fermandosi prima se si entra in contatto con un ostacolo.

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.

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 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 Visitare e contare tutte le casello non-muro accessibili a Quadretto.

3.22 Analizzare tutte le possibili situazioni in cui si può trovare Quadretto nell'algoritmo 1 e verificare che in ogni caso il robot riesce a togliersi dall'ostacolo.

3.23 Realizzare un sensore virtuale *distanza* che ritorna la distanza dall'ostacolo posto davanti a Quadretto.

3.24 Realizzare un sensore virtuale *contatto* che ritorna `TRUE` se e solo se Quadretto è a contatto con un ostacolo con uno dei suoi 4 lati.

3.25 Realizzare un sensore virtuale di distanza che rileva la distanza dall'ostacolo posto davanti. Suggerimento: avanzare fino ad arrivare a contatto con il muro, contando i passi, e poi ritornare nella situazione di partenza.

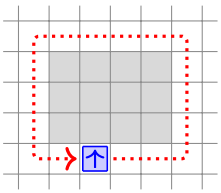
3.26 Adattare l'algoritmo 18 (*chiudi stanza*) in modo da gestire le situazioni in cui la stanza abbia una o entrambe le dimensioni pari ad una casella.

3.27 Stabilire l'effetto dell'algoritmo 22 (*striscia*) nel caso in cui non sia vera la precondizione *tocca*(`LEFT`).

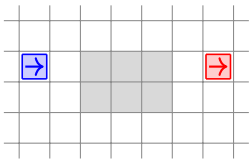
3.28 Stabilire l'effetto dell'algoritmo 25 (*costeggia*) se non ci sono ostacoli o gli ostacoli sono attaccati al muro perimetrale.

3.29 Stabilire l'effetto dell'algoritmo 37 (*esci dal labirinto*) nel caso in cui la scacchiera sia completamente libera da ostacoli.

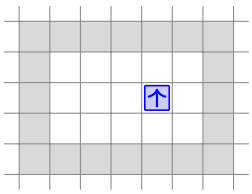
3.30 Quadretto si trova a contatto con un muro rettangolare. Far circumnavigare Quadretto attorno al muro, a contatto con il muro sul lato sinistro, fino a tornare al punto di partenza.



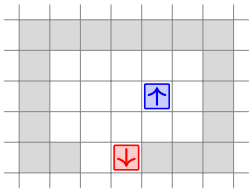
3.31 Quadretto si trova davanti ad un muro rettangolare. Portare Quadretto dall'altra parte del muro, in una posizione speculare al muro rispetto alla posizione di partenza.



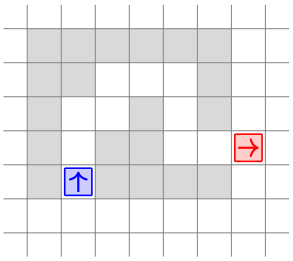
3.32 Quadretto si trova all'interno di un recinto rettangolare. Determinare il perimetro (lunghezza del perimetro interno) e l'area (numero di celle all'interno del recinto).



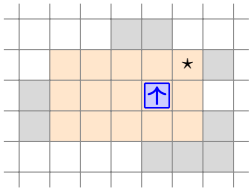
3.33 Quadretto si trova all'interno di una stanza rettangolare di dimensioni $m \times n$ unità ed avente su un lato un'apertura di 1 o più unità. Far uscire Quadretto dalla stanza, portandolo sul bordo. Analizzare e risolvere il problema nel caso in cui non si conoscano le dimensioni della stanza.



3.34 Quadretto si trova all'interno di un tunnel lineare di larghezza unitaria. Farlo uscire dal tunnel.



3.35 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.



3.36 Quadretto si trova all'interno di una stanza rettangolare chiusa su tutti quattro i lati. All'interno della stanza è presente un ostacolo di forma generica, non appoggiato alle pareti. Individuare l'ostacolo, portando Quadretto a contatto con l'ostacolo. Analizzare e risolvere la situazione nel caso in cui la stanza sia completamente sgombra da ostacoli, fermando Quadretto nel momento in cui si sia riconosciuto che nella stanza non è presente alcun ostacolo. 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 alle pareti.

3.37 Nell'ambiente di Quadretto ci sono alcune caselle colorate con diversi colori (non ci sono caselle-muro); contare le caselle di un dato colore.

3.38 All'interno della scacchiera, staccato dal muro perimetrale, c'è un ostacolo connesso e circumnavigabile.

1. Calcolare la lunghezza del perimetro esterno dell'ostacolo.
2. Calcolare l'area (numero di caselle) dell'ostacolo.

3.39 All'interno della scacchiera ci sono degli ostacoli 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 ostacoli presenti.

3.40 Raggiungere una data posizione, evitando eventuali ostacoli presenti sulla scacchiera.

3.41 Sulla scacchiera è presente un solo blocco, staccato dal bordo. Spingerlo:

1. contro il bordo
2. in un angolo
3. nell'angolo più vicino

3.42 Sulla scacchiera sono presenti due blocchi, non entrambi in un angolo. Spingerli in modo da farli toccare fra loro su un lato.

3.43 Sulla scacchiera sono presenti quattro blocchi separati dal bordo e separati fra loro. Spingerli in modo da raggrupparli in modo da formare un quadrato (di lato 2 caselle).

APPENDICE A

QUADRETTO

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

QUADRETTO è un elementare ambiente di *robotica educativa* caratterizzato da un automa virtuale, chiamato *Quadretto* (nel seguito semplicemente Q), che si muove su una scacchiera quadrata visualizzata sul video. In ogni istante Q occupa esattamente una casella della scacchiera, ha un verso di avanzamento rispetto al quale può avanzare alla casella successiva e ruotare ad angolo retto, a sinistra o a destra. Q può essere movimentato mediante dei comandi diretti attivabili mediante bottoni presenti su video oppure mediante analoghi comandi del linguaggio `LFR` (come descritto nel cap. *Muovere un robot*) oppure Python (come descritto nel cap. *Interagire con l'ambiente*).

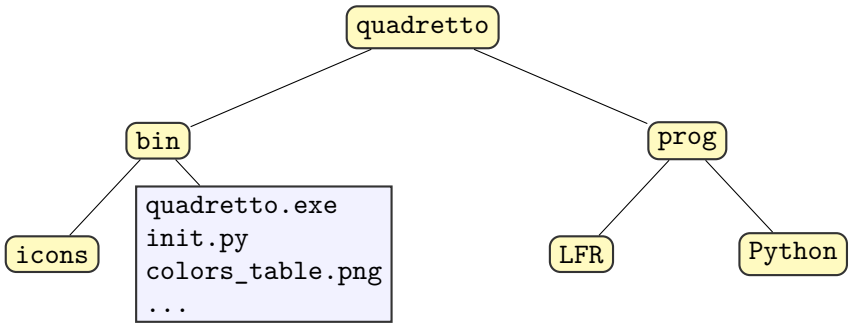
Attenzione: questa è una versione provvisoria ed incompleta del manuale; contiene sicuramente errori e ci sono delle funzionalità descritte che non sono perfettamente allineate con quanto effettivamente realizzato.

A.1 Installazione

Per installare QUADRETTO si può procedere come segue:

1. scaricare il file `quadretto.zip` da `github.com/algmath/quadretto`
2. decomprimere il file `quadretto.zip` in una cartella (in una generica posizione)

Si ottiene la seguente struttura di cartelle:



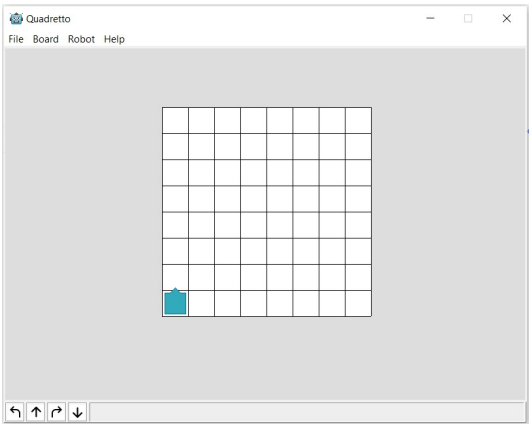
La cartella `bin` contiene il programma; in particolare: il file `quadretto.exe` è il programma che attiva la finestra principale dell'applicazione; il file `init.py` viene automaticamente eseguito inizialmente e contiene le istruzioni (in linguaggio Python) di configurazione dell'ambiente; le principali sono descritte a seguire:

- `setviewport(`*xmin,ymin,xmax,ymax*`)`: setta la posizione su video e la grandezza della finestra dell'applicazione (in unità pixel)
- `setboard(`*nc,np*`)`: setta il numero *nc* di caselle per lato della scacchiera ed il numero *np* di pixel di ciascuna casella della scacchiera
- `setprgpath(`*path*`)`: setta il percorso da dove caricare i programmi
- `setlibpath(`*path*`)`: setta il percorso delle librerie, da dove importare i moduli e le classi Python
- `setcolor(WALL,`*colore*`)`: setta il colore del muro; analogamente per `EMPTY`, `LEFT`, `RIGHT`, ... (*colore* è il nome identificativo del colore secondo le convenzioni del modulo *tkinter* di Python, riportate nel file `colors_table.png`)

Nella cartella `prog` vengono memorizzati i programmi utente; in particolare nella sotto cartella `prog\LFR` i programmi in linguaggio `LFR` e nella sotto cartella `prog\python` quelli scritti in linguaggio Python; la cartella `prog\Python\lib` contiene i moduli Python che vengono importati nei programmi scritti dall'utente.

A.2 L'interfaccia grafica

QUADRETTO si attiva mediante l'esecuzione del programma `quadretto.exe` presente nella sotto cartella `bin`. Al momento dell'esecuzione viene eseguito il file `init.py` presente nella stessa cartella; tale file contiene alcune istruzioni (in linguaggio Python) di configurazione dell'ambiente. Si presenta la seguente finestra.



Nell'ambiente di questa finestra sono predisposte le varie modalità di manipolazione di Q:

- attivare le voci di menù corrispondenti alle seguenti funzionalità:

  File > *Edit new* : scrittura di un nuovo programma
  File > *Edit ...* : scrittura del programma che viene selezionato
  File > *Exit* : termina il programma QUADRETTO
  Board > *Empty* : elimina gli oggetti sulla scacchiera (blocchi e muri)
  Board > *Reset* : riporta la scacchiera allo stato iniziale
  Board > *Load ...* : carica un file di configurazione della scacchiera
  Robot > *Home* : riporta Q alla casella $(1, 1)$, rivolto a Nord
  Robot > *Import ...* : seleziona ed importa un modulo
  Robot > *Start* : fa partire Q (si ferma quando incontra un ostacolo)
  Robot > *Run ...* : seleziona ed esegue un programma
  Robot > *Rerun* : esegue l'ultimo programma eseguito
  Robot > *Trace* : colora le caselle sulle quali transita Q
  Robot > *Stop* : interrompe l'esecuzione del programma e ferma Q
  Help > *About* : visualizza la versione del programma QUADRETTO
  Help > *Manual* : visualizza il manuale del programma QUADRETTO

- bottoni di movimento: sono i 4 bottoni con freccia posti in basso a sinistra; servono per muovere Q "manualmente"
- linea di programmazione: è la linea posta in basso, sulla destra dei bottoni di movimento, dove si scrivono, in linguaggio `LFR`, singoli comandi o programmi da eseguire

A.3 Programmazione

Q può essere gestito e programmato in varie modalità.

Azionamento diretto

Q può essere movimentato direttamente, mediante i 4 bottoni di movimento:

↑ : avanzamento di una casella

↓ : indietreggiamento di una casella

↶ : rotazione a sinistra

↷ : rotazione a destra

In alternativa ai bottoni di movimento si possono utilizzare i tasti freccia, con gli usuali significati.

Esempio A.3.1 - La pressione in sequenza dei bottoni ↑ ↑ ↶ ↑ ha l'effetto di far eseguire a Q un percorso ad "elle", come un cavallo degli scacchi. □

Esecuzione di un programma

Q può essere comandato mediante un programma di movimento scritto nel campo dei comandi (posto a lato delle icone di movimento diretto). Si entra in questa modalità con un clic sulla linea dei comandi e si esce con il tasto *Escape*. Una volta entrato in modo comando, viene disabilitato il movimento di Q mediante i tasti freccia. Il programma viene scritto mediante i seguenti caratteri:

`F` : avanzamento alla casella davanti

`B` : indietreggiamento di una casella

`L` : rotazione a sinistra

`R` : rotazione a destra

Si possono comporre questi comandi elementari mediante i seguenti controlli, creando

$n\ \alpha$: ripete n volte il programma α

`[` $\alpha_1\,\alpha_2\,\ldots\,\alpha_n$ `]` : raggruppa la sequenza α

`-`α : inversione del programma α

Viene così definito un vero e proprio linguaggio per la programmazione di Q (denominato *linguaggio* `LFR`), la cui sintassi e semantica è descritta nel cap. *Muovere*. Una volta scritto il programma, mediante il tasto di *Invio* il programma viene eseguito; eventuali errori di sintassi vengono segnalati mediante l'apertura di una apposita finestra di messaggio. Una modalità alternativa consiste nello scrivere il programma in un file testo (con estensione `.lfr`) e poi eseguirlo mediante la voce di menu *Robot* > *Run ...* con la quale si seleziona il programma da eseguire.

Esempio A.3.2 - Il programma `4[2FR]` fa percorrere a Q un cammino quadrato di lato 3. □

Programmazione in Python

La modalità più duttile per programmare i movimenti di Q consiste nello scrivere e poi eseguire un programma in linguaggio Python: dapprima, in un file sorgente si scrive, mediante un qualsiasi text-editor,il programma Python; successivamente mediante la voce di menù *Run* > *Exec program ...* si seleziona il programma da eseguire.

Le istruzioni Python per muovere Q sono:

- `forward()`: avanza alla casella successiva
- `back()`: indietreggia di una casella
- `left()`: ruota a sinistra
- `right()`: ruota a sinistra
- `posx()`: coordinata orizzontale di Q
- `posy()`: coordinata verticale di Q
- `versus()`: verso di avanzamento di Q (`NORTH`, `WEST`, `SOUTH`, `EAST`)

Esempio A.3.3 - Il programma che segue fa compiere a Q un cammino lungo un quadrato di lato 3.

```
for m in range(4):
    for n in range(2):
        forward()
    right()
```

□

Altri comandi in Python saranno presentati nei prossimi paragrafi.

A.4 Oggetti sulla scacchiera

Sulle caselle all'interno della scacchiera è possibile disporre degli oggetti, di due diverse tipologie: *muri* e *blocchi*. Questi oggetti vengono creati selezionando la voce *Wall* o la voce *Block* dal menu popup che si apre con clic con il tasto destro del mouse sulla casella desiderata. Ogni blocco è caratterizzato da un numero progressivo che appare sul blocco stesso.

I muri si comportano come la parte esterna alla scacchiera e vincolano il movimento di Q; i blocchi, invece, sono degli oggetti che possono essere spinti da Q quando sono a suo diretto contatto; la spinta causata dall'avanzamento di Q 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 A.1) alla fine della fila di blocchi posta a diretto contatto con Q.

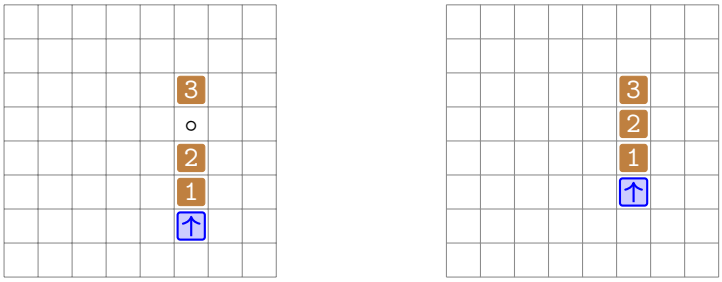


Figura A.1: Situazione prima e dopo la spinta di Quadretto.

La spinta può avvenire anche con una spinta in indietreggiamento di Q. 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.

La gestione del movimento in presenza di oggetti viene gestita mediante i seguenti sensori:

- `front()`: sensore di tatto che ritorna uno dei valori `WALL`, `BLOCK`, `SPACE` a seconda che nella casella davanti ci sia un muro, un blocco o spazio
- `touch()`: sensore di contatto che ritorna *True* se davanti c'è un muro o un blocco; equivale a `front() in {BLOCK, WALL}`
- `blocked()`: sensore di stato che ritorna *True* se l'ultima azione di movimento (`forward()` o `backward()`) non è riuscita a spostare Q a causa di un muro o da dei blocchi che non possono essere spinti in avanti
- `dist()`: sensore di distanza che ritorna il numero di caselle davanti libere da oggetti

Esempio A.4.1 - Il programma che segue avanza Q fino ad arrivare ad una situazione di blocco.

```
while not blocked():
    forward()
```

□

A.5 I colori delle caselle

Le caselle della scacchiera possono essere colorate. I possibili colori delle caselle ed il loro significato per Q è descritto a seguire:

- `EMPTY`: casella neutra (senza indicazione di comando)
- `LEFT`: rotazione a sinistra
- `RIGHT`: rotazione a destra
- `INVERT`: inverte il senso di marcia
- `BLACK`: casella per il line-follower

Q percepisce il colore della casella davanti mediante il sensore

- `color()`: colore della casella davanti

ed è in grado di lasciare una traccia dove passa se viene attivata l'opzione

- `trace(TRUE)`: traccia una striscia nera sulle caselle in cui transita

La colorazione delle caselle può essere fatta interattivamente con clic-dx sulla casella desiderata: si apre un menu popup mediante il quale è possibile selezionare il colore desiderato. Una volta selezionato un colore è sufficiente un semplice clic su una casella per colorarla con il colore selezionato precedentemente; un clic-sx su una casella colorata ha l'effetto di togliere il colore. In alternativa alla modalità interattiva sopra descritta, è possibile colorare una casella di coordinate (r, c) con l'istruzione

- `fill(`r`,`c`,`*col*`)`: riempie la casella (r, c) con il colore *col*

Coding-on-board

È una modalità di programmazione dei movimenti specificando il comportamento di Q quando arriva in una data casella: all'arrivo nella casella 🔼 esegue l'azione (di rotazione) in base al colore della casella: se è *gialla* ruota a *sinistra*, se *rossa* ruota a *destra* (nel caso in cui sia *bianca* non esegue alcuna rotazione). Svolta l'eventuale rotazione, Q fa un passo in avanti nella direzione corrente. Il movimento viene attivato mediante la voce di menù *Robot* > *Start* che fa avanzare Q (ed eseguire l'eventuale azione di rotazione). Con questa modalità, anziché programmare il robot, si programma la struttura dello spazio definendone la struttura in modo che il robot che vi transiterà segua le direttive di movimento specificate in ciascuna casella.

BIBLIOGRAFIA

1. Alessandro Bogliolo, *Coding in Your Classroom, Now!*, Giunti Scuola, 2018.
2. George A. Bekey, *Autonomous Robots*, The MIT Press, 2005.
3. Seymour Papert, *Mindstorms: Bambini computers e creatività*, EMME Edizioni, 1984.
4. Seymour Papert, *The Children's Machine: Rethinking School in the Age of the Computer*, Basic Books, 1993.