Capitolo 8

Un modello di computazione per il calcolatore

Quando si studia la realtà fisica che ci circonda, seguendo l'approccio scientifico Galileiano, si tenta di condensare all'interno di un *modello* quelle che sono le evidenze sperimentali del particolare sottosistema che si sta analizzando. Un modello è una rappresentazione astratta del sistema, che ne riproduce le caratteristiche, le peculiarità e i comportamenti fondamentali. La definizione del modello va fatta solitamente ammettendo alcune ipotesi generali sul sistema e deducendo le leggi matematiche che sono alla base della descrizione del suo comportamento. Quando si dispone di un modello è possibile fare delle *previsioni* sul comportamento del sistema, sfruttando le equazioni che lo rappresentano.
Per esempio il *modello del corpo rigido*, impiegato nella *meccanica classica*, consente di descrivere il comportamento di un corpo caratterizzato dall'ipotesi che non sia deformabile. Ciò significa che, scelti due punti qualunque x_1, y_1, z_1 e x_2, y_2, z_2 appartenenti al corpo e detta d_{12} la loro distanza iniziale, il vincolo di rigidità è analiticamente espresso dalla relazione:

$$(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2 - d_{12} = 0$$

che deve valere per ogni istante successivo a quello iniziale. Del corpo rigido possiamo studiare la sua *cinematica* e la sua *dinamica*, deducendo delle equazioni che ci consentono di fare delle previsioni sul comportamento del sistema. Per esempio è possibile stabilire il punto esatto in cui una sfera indeformabile di ferro, di diametro di 1 cm, cadrà quando lanciata verso l'alto con un'inclinazione di 32° rispetto al piano orizzontale e con una velocità iniziale di $3,5$ m/s.
Qualunque modello offre tuttavia una rappresentazione solo parziale della realtà, poiché tende a evidenziare le sole grandezze fisiche che ci interessano al livello di analisi in cui stiamo operando; esso è legato, in modo indissolubile, alle ipotesi che facciamo sul sistema. Se le ipotesi diventano meno vincolanti, il modello deve essere "esteso" per poter tener conto del nuovo stato di cose. Se per esempio prendiamo la nostra sferetta di ferro e ipotizziamo che si sposti a velocità molto elevate, prossime alla velocità della luce, allora il modello delle meccanica classica non è più sufficiente a descrivere il sistema poiché, per esempio, il diametro della sfera subirà una contrazione nella direzione del moto descritta dalla famosa legge di *Lorenz-FitzGerald*

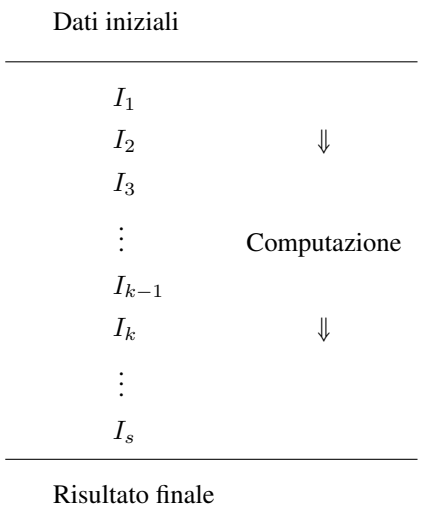
$$D_v = D\sqrt{1 - (v/c)^2}$$

dove D_v è il diametro nella direzione dello spostamento a velocità v, c è la velocità della luce e D è il diametro della sfera in quiete. Il modello deve dunque essere esteso per tener conto delle equazioni della *relatività ristretta*, che offrono una descrizione più accurata del sistema. Assumendo il punto di vista della *Relatività ristretta* possiamo allora affermare che il modello della meccanica classica offre una buona rappresentazione della realtà quando le

velocità in gioco sono trascurabili rispetto alla velocità della luce.
Sulla base di quanto detto finora sembra ragionevole poter costruire un *modello* anche per il calcolatore descritto nel capitolo 7, che specifichi con precisione le ipotesi che riguardano il funzionamento del sistema, in modo che si possano fare delle *previsioni* sul suo comportamento; per esempio sarebbe interessante individuare quali problemi si possono effettivamente risolvere col metodo algoritmico-procedurale anticipato nel paragrafo 1.4 e quali no, oppure conoscere i tempi necessari per ottenere una soluzione di un problema risolubile. In questo capitolo ci occuperemo proprio della costruzione di un modello di computazione per il calcolatore. Come vedremo di modelli ce ne sono tanti, ma si potrebbe dimostrare che essi sono tra loro tutti *equivalenti*, nel senso che portano tutti allo stesso insieme di problemi risolubili o, per esprimersi in modo un po' più tecnico, allo stesso insieme di *funzioni computabili*.
Il modello che sceglieremo è il *modello RAM*, che al contrario degli altri ha il pregio di essere basato sulla struttura architetturale dei computer. Grazie al modello RAM saremo in grado di capire la vera natura della computazione, quali sono i problemi effettivamente risolubili da un calcolatore e quali non lo sono e quali sono i tempi necessari alla risoluzione, in funzione della dimensione del problema.

8.1 Il concetto di algoritmo

Un calcolatore è un dispositivo che serve per eseguire *programmi*. La loro esecuzione è finalizzata alla risoluzione di *problemi* del mondo reale anche molto complessi ed eterogenei tra loro. La *codifica simbolica*, cioè la possibilità di associare un qualunque significato a un simbolo, coniugata con la possibilità offerta dal calcolatore di manipolare i simboli in modo logicamente strutturato, consente di trasformare i problemi del mondo reale in *problemi astratti*, di natura logico-simbolica e descrivibili con gli strumenti della matematica. Il problema astratto viene poi risolto in un tempo finito, avvalendosi dell'approccio *procedurale* o *algoritmico* descritto nella sezione 1.4 e che richiamiamo brevemente: si parte da un insieme di informazioni fornite al sistema di elaborazione dall'esterno, i cosiddetti *dati iniziali*, che vengono successivamente elaborati secondo una procedura ordinata a passi, descrivibile in modo preciso ed esauriente come una sequenza *finita* $I_1, I_2, \ldots, I_s$ di *istruzioni elementari*. Il risultato dell'esecuzione dell'istruzione I_k dipende, oltre che dal tipo di istruzione, anche dai dati d'ingresso al passo k, che sono costituiti da eventuali dati esterni e dai dati che derivano dall'esecuzione dell'istruzione I_{k-1} del passo precedente.



Alla procedura in questione viene attribuito il nome di *algoritmo*; la sua esplicitazione rigorosa in un linguaggio comprensibile alla macchina viene chiamata *programma* $\mathcal{P}$, mentre si riserva il termine di *computazione* al processo che consiste nell'esecuzione dell'algoritmo (del programma) a partire dai dati iniziali. Il risultato della

computazione, cioè i dati generati in uscita al termine della stessa e che devono essere prodotti in tempo finito, consente di esprimere in modo codificato la soluzione al problema del mondo reale.

L'approccio procedurale-algoritmico non è l'unico che si possa concepire per risolvere problemi di natura logico-simbolica; esistono infatti anche altri *paradigmi* di computazione, quali ad esempio le *reti neuronali*, gli *algoritmi genetici*, la *computazione DNA* e la *computazione quantistica*; tuttavia il metodo procedurale è l'unico basato su un solido *modello di computazione*, legato ai lavori di Church, Turing, Gödel e Kleene, sui quali si è poi sviluppata tutta la *teoria della computazione*; essa stabilisce, tramite rigorosi teoremi matematici, i limiti intrinseci dell'approcio procedurale-algoritmico, distinguendo tra *problemi risolubili* (o *predicati decidibili*) e *problemi non-risolubili* (o *predicati indecidibili*). La successiva *teoria della complessità computazionale* si occupa invece di distinguere, all'interno dei problemi risolubili, i gradi di complessità degli stessi, legati al numero di passi elementari che bisogna svolgere per ottenere la soluzione del problema. In quest'ambito la distinzione è tra problemi *trattabili* in un tempo accettabile (in *tempo polinomiale*) e problemi di fatto *intrattabili*, (almeno quando le dimensioni del problema è sufficientemente grande), poiché richiederebbero un numero *esponenziale* di passi, portando a dei tempi di attesa per la soluzione assolutamente inaccettabili (p.es. $3,2$ milioni di anni).

Nello schema procedurale si ipotizza che le istruzioni elementari siano *immediatamente eseguibili*; l'idea di base, molto comune anche nella pratica quotidiana, è che per risolvere un problema complesso sia necessario attuare una strategia la cui descrizione venga specificata da un certo numero di *passi elementari*. Se per esempio voglio uscire dall'ufficio per andare a prendere l'autobus, dovrò aprire la porta se questa è chiusa, scegliere se andare a destra o a sinistra per raggiungere l'uscita dell'edificio, percorrere il corridoio sino alla scalinata, scegliere se salire o scendere dalla stessa ecc. Immaginiamo ora che mi trovi in un edificio che non conosco, e che per raggiungere la fermata abbia in mano un foglio con le istruzioni sul percorso; anche se non ho idea di come sia fatto l'edificio è sufficiente che esegua alla lettera l'elenco delle istruzioni per arrivare alla fermata. Le istruzioni elementari (apri la porta, gira a destra, scendi le scale...) devono però essere alla mia portata e immediatamente eseguibili, senza l'ausilio di ulteriori "istruzioni" supplementari.

Se caliamo questo ragionamento nell'ambito dei calcolatori e se teniamo conto che essi sfruttano le tecnologie elettroniche, l'immediata eseguibilità di un'istruzione implica un'attività a livello circuitale realizzata da un *agente di calcolo* $\mathcal{A}$. Le operazioni eseguite da questo dispositivo sono molto semplici, e possono riguardare una gestione dell'informazione a livello di codice ASCII (nel caso si tratti di manipolazione di simboli su tale alfabeto) oppure di blocchi di *byte* che rappresentano dei numeri seconda una delle notazioni usate (complemento a 2, *floating point*,...) o anche una lettura o scrittura dei dati in una memoria ecc. Di conseguenza le capacità di elaborazione di $\mathcal{A}$ sono necessariamente limitate.

Per l'esecuzione dell'istruzione da parte dell'agente di calcolo potrebbe essere necessaria una *memoria* $\mathcal{M}$ di supporto (memoria RAM), per esempio per memorizzare risultati intermedi di una computazione, che può essere arbitrariamente grande.

L'interazione tra l'agente di calcolo $\mathcal{A}$ e il programma $\mathcal{P}$ avviene sulla trama di un *tempo discreto*, che viene specificato da un orologio interno del calcolatore (*clock*). Il tempo discreto corrisponde a una cadenza temporale prefissata e costante (p.es. 1 miliardesimo di secondo) in corrispondenza della quale possono essere effettuate le operazioni elementari a carico delle circuiteria. In altre parole il sistema rimane "congelato" tra due istanti di tempo discreto successivi. Si osservi che la modalità di interazione in tempo discreto è alternativa a una modalità in *tempo continuo*, che era invece prerogativa dei vecchi sistemi analogici, nei quali le varie grandezze variano con continuià nel tempo.

Un'altra ipotesi implicita che si fa nella realizzazione di un calcolatore è che l'interazione tra l'agente di calcolo $\mathcal{A}$ e il programma $\mathcal{P}$ sia *deterministica*. Ciò significa che, a partire dallo stesso insieme di dati iniziali, l'esecuzione di un insieme specificato di istruzioni (programma) porta sempre allo stesso risultato finale. Tale modalità si contrappone a una computazione nella quale esistano dei meccanismi *aleatori*, in cui sia possibile scegliere tra più di un percorso per la computazione a partire dalle stesse condizioni iniziali.

Per quanto riguarda invece la natura delle istruzioni $I_1, I_2, \ldots, I_s$, possiamo osservare che ogni istruzione I_j deve appartenere a un certo insieme $\mathcal{I} = \{I_1, I_2, \ldots l_k\}$ di istruzioni elementari che la circuiteria di $\mathcal{A}$ è in grado di svolgere. Nella logica dell'approccio procedurale le informazioni elementari, proprio in quanto tali, devono essere necessariamente "poche". Come abbiamo visto, la struttura architetturale acquisita dai vari sistemi porta a distinguere tra due filosofie, quelle $CISC$, che sta per *Complex Instruction Set Computer*, e quella $RISC$, che significa *Reduced Instruction Set Computer*. Nel primo caso le istruzioni dette elementari consentono di svolgere

operazioni che sono comunque relativamente complesse e articolate, come la lettura di un dato in memoria, la sua modifica e il suo salvataggio direttamente in memoria tramite una singola istruzione. Si è però osservato che questa impostazione risulta poco vantaggiosa, poiché offre in genere costi maggiori e prestazioni modeste, visto che i tempi di decodifica e di esecuzione sono in generale maggiori anche per le istruzioni più semplici. Nel caso dell'architettura $RISC$ c'è invece un insieme molto ridotto di istruzioni effettivamente elementari (lettura e scrittura in memoria, copia di un dato, somma, sottrazione ecc.), e ciò consente di far lavorare i processori in modo più efficiente. A prescindere dal fatto che si usi un insieme $CISC$ o $RISC$, è ovvio che tale insieme debba avere una dimensione finita, anche se non è però necessario specificare una *limitazione superiore* per k.
Un altro elemento importante da considerare tra le ipotesi del calcolo procedurale-algoritmico è relativa alla dimensione dei dati di ingresso; benché non abbia senso porre alcuna limitazione ad essa, è tuttavia doveroso considerarla come una quantità finita; lo stesso può dirsi anche per la lunghezza della computazione: non si può porre un limite al numero di istruzioni che vengono eseguite prima di arrivare alla fine del programma, visto che tale numero potrebbe anche essere (molto) maggiore di n nel caso il programma contempli dei *cicli*. Tuttavia, affinché il sistema funzioni bene per le finalità per le quali esso è stato costruito e segua lo spirito della definizione di algoritmo, sarebbe auspicabile che la computazione durasse un tempo finito, anche se non limitabile. Come vedremo, però, saremo costretti ad accettare nel modello i casi in cui la computazione incappa in un ciclo infinito o *loop*, dal quale non si può uscire. In tal caso la macchina continua a girare all'infinito, senza pervenire mai all'esecuzione di un'istruzione che porti a uno stop; l'utente interpreta questo fatto come un blocco della computazione e deve forzare dall'esterno l'uscita dal programma oppure, nei casi peggiori, riavviare la macchina, perdendo in entrambi i casi tutto il lavoro svolto. Possiamo allora individuare il seguente elenco delle caratteristiche associate alla nozione informale di algoritmo:

Nozione informale di algoritmo

1)	insieme di istruzioni $I_1, I_2, \ldots, I_s$ di lunghezza finita	$\mathcal{P}$ (*programma*)
2)	c'è un *agente di calcolo*	$\mathcal{A}$ (circuiteria)
3)	c'è a disposizione della *memoria*	$\mathcal{M}$ (memoria RAM)
4)	$\mathcal{A}$ interagisce con $\mathcal{P}$ in *modalità discreta*	(macchina discreta)
5)	$\mathcal{A}$ interagisce con $\mathcal{P}$ in *modalità deterministica*	(macchina deterministica)
	Bisogna fissare un limite finito:	
6)	sulla dimensione dei dati d'ingresso?	NO
7)	sulla dimensione dell'insieme $\mathcal{I} = \{I_1, I_2, \ldots l_k\}$ di istruzioni?	NO
8)	sulla dimensione della memoria?	NO
9)	sulla capacità di computazione di A?	SÌ
10)	sulla lunghezza della computazione?	NO
	Tuttavia:	
11)	sono ammesse computazioni con un numero infinito di passi (computazioni non terminanti)	

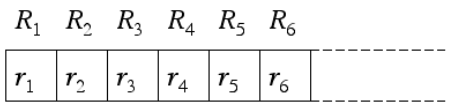
ALGORITMO + 11) = PROGRAMMA

All'algoritmo descritto dai punti $1 - 10$ siamo dunque costretti ad aggiungere l'ipotesi di una computazione in qualche caso non terminante; ciò che ne esce rappresenta la proiezione del concetto astratto di algoritmo, che porta sempre a una soluzione in un tempo finito, sulla realtà del modello effettivo di computazione, che in certe sfortunate situazioni porta a un programma $\mathcal{P}$ che cicla all'infinito. Senza entrare nei dettagli del problema relativo al punto 11) possiamo dire solo che, se lo escludessimo pretendendo di avere a che fare con un modello di calcolo basato sulle sole computazioni terminanti, allora ci si troverebbe in una situazione in cui esisterebbero dei problemi palesemente risolubili mediante approccio procedurale, ma che *non* potrebbero essere risolti all'interno del nostro modello.

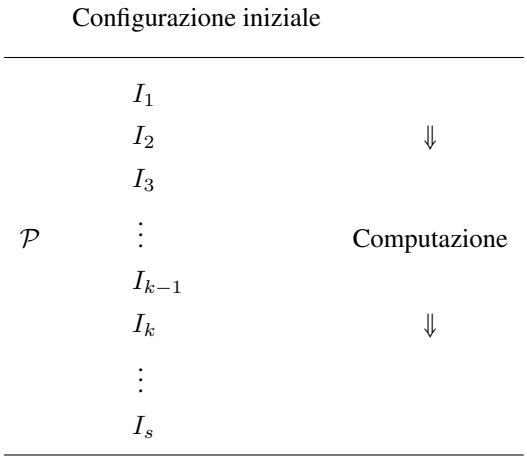
8.2 Il modello RAM

Nel paragrafo 1.3 abbiamo visto che l'informatica nacque sul solco delle riflessioni inerenti gli aspetti logico-fondazionali della Matematica, centrati sulla risoluzione del famoso *Entscheidungsproblem* (Problema della decisione). La sua risoluzione, in senso negativo, da parte di *Alonzo Church* e di *Alan Turing* nel 1936, costituisce il punto di partenza della moderna *Teoria della Computabilità*. Tuttavia il lavoro di Church era molto astratto e di difficile comprensione, mentre il modello della *Macchina di Turing* fece ben presto breccia e divenne in breve tempo il modello di riferimento. Ciononostante, questi approcci alla computazione erano stati creati non per rispondere alle esigenze di modellare il funzionamento di un *computer*, che all'epoca non esisteva ancora, ma per dare soluzione a un problema della logica. Ci si trovò dunque, ben presto, nella spiacevole situazione di avere dei *computer* pienamente funzionanti, associati però a dei modelli di computazione che non avevano nulla a che fare con le linee architetturali del sistema che intendevano rappresentare. Oltre ai due modelli citati se ne introdussero successivamente anche altri (Gödel-Kleene, Gödel-Herbrand-Kleene, Post, Markov), tutti però basati su procedimenti di calcolo molto astratti, che non trovavano rispondenza diretta con quanto attuato dai *computer* reali. Solo nel 1963 Shepherdson e Sturgis introdussero un modello, che chiameremo *modello RAM*, che prendeva spunto dalla effettiva struttura del calcolatore moderno, basata come abbiamo visto sull'architettura di Von Neumann e sulla memoria RAM. Passiamo ora alla descrizione di tale modello.

Il modello RAM è basato su un *nastro di memoria di lunghezza infinita*, che rappresenta una idealizzazione delle memorie del calcolatore, e da un *insieme di istruzioni*, che rappresentano le istruzioni in linguaggio macchina (si veda la figura 8.1a). La memoria è costituita da celle che contengono dei numeri naturali r_i, e dunque $r_i \in \mathbb{N}$. In questo senso c'è una differenza concettuale con i computer reali, che come noto lavorano sulla base di un alfabeto binario; tuttavia tale differenza non è rilevante, poiché una qualunque stringa binaria può essere interpretata come numero intero e qualunque numero intero può avere una rappresentazione binaria. L'aspetto concettualmente rilevante è, semmai, il fatto che ogni cella, contenendo un numero intero non limitabile superiormente, è associata a una quantità d'informazione a sua volta non limitabile.



(a) Il nastro di memoria del modello RAM



(b) Il programma $\mathcal{P}$ associato al nastro

Figura 8.1: Gli elementi del modello RAM

Il contenuto delle celle può essere modificato da delle *istruzioni* previste dal modello, che sono state scelte in modo tale da costituire un insieme in qualche senso minimo. Tali istruzioni sono impiegate per realizzare un *programma* $\mathcal{P} = I_1, I_2, ..., I_s$ (fig. 8.1b), che a partire da una *configurazione iniziale* del nastro porta a una *configurazione finale*, dalla quale si ricava il *risultato* della *computazione*; quest'ultima corrisponde all'esecuzione, passo passo, delle istruzioni del programma.

Analizziamo ora le istruzioni, che sono solamente le seguenti quattro:

Istruzione di azzeramento - $Z(n)$ comporta l'azzeramento del contenuto della cella R_n.
Esempio: Se applico l'istruzione $Z(2)$ al nastro della prima riga di figura 8.2 azzero il contenuto della seconda cella (fig.8.2a)

Istruzione di incremento - $S(n)$ comporta l'incremento del contenuto della cella R_n di un'unità.
Esempio: Se applico l'istruzione $S(5)$ al nastro della seconda riga di figura 8.2 incremento di 1 il contenuto della quinta cella (fig.8.2b).

Istruzione di trasferimento - $T(m,n)$ comporta la copia del contenuto della cella R_m nella cella R_n.
Esempio: Se applico l'istruzione $T(2,4)$ al nastro della terza riga di figura 8.2 copio il contenuto della seconda cella nella quarta (fig.8.2c).

Le prime tre istruzioni si chiamano *aritmetiche*, poiché operano manipolando numeri naturali. Usando solamente questo tipo di istruzioni non sarebbe però possibile risolvere problemi con un minimo di interesse pratico, poiché nella vita reale ci si trova sempre nella situazione di dover scegliere strategie diverse a seconda che alcune condizioni siano o meno soddisfatte. In altre parole è necessario introdurre un'istruzione *logica* che ci consenta di aprire percorsi diversi alla computazione.

Istruzione di salto condizionato - $C(m,n,q)$ L'esecuzione di tale istruzione implica il controllo del contenuto delle celle R_m e R_n; se $r_m = r_n$ l'esecuzione del programma continua con l'istruzione I_q, altrimenti si continua con l'istruzione successiva.
Esempio: Se applico l'istruzione $C(1,6,9)$ al nastro della quarta riga di figura 8.2, poiché le celle 1 e 6 hanno lo stesso contenuto, la computazione continua con l'istruzione I_9 del programma (fig.8.2c).

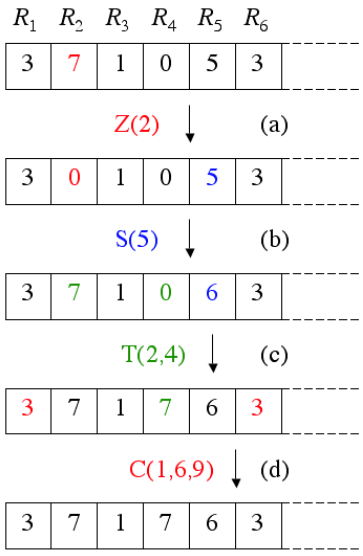


Figura 8.2: Applicazione di alcune istruzioni al nastro di memoria

Vediamo di delineare in modo più preciso i dettagli della computazione in modo che, a partire dal programma e dalla configurazione iniziale, si sappia con precisione quali sono i passi da eseguire e come si ottiene il risultato finale. Lo facciamo evidenziando alcuni punti chiave, che vengono di seguito illustrati.

Computazione - Per eseguire la computazione RAM bisogna fornire un *nastro*, caricato con una *configurazione iniziale* costituita da una sequenza $a_1, a_2, a_3, ...$ di numeri naturali; se $\mathcal{P} = I_1, I_2, ..., I_s$ è il programma

associato alla macchina, la computazione inizia eseguendo l'istruzione I_1. Dopo averla eseguita la macchina RAM dovrà eseguire la *prossima istruzione* (I_{NEXT}) finché si raggiunge uno STOP (se mai si raggiunge).

Prossima istruzione - Se I_k è l'istruzione corrente, la prossima istruzione da eseguire I_{NEXT} è così definita:

- Se I_k è un'istruzione aritmetica $I_{NEXT} = I_{k+1}$ (fig. 8.3a)
- Se $I_k = C(m, n, q)$ si ha $I_{NEXT} = \begin{cases} I_q & \text{se } r_m = r_n \quad \text{(fig. 8.3b)} \\ I_{k+1} & \text{se } r_m \neq r_n \quad \text{(fig. 8.3c)} \end{cases}$

La computazione ha un flusso regolare quando si ha a che fare con istruzioni aritmetiche, oppure quando si incontra un'istruzione di salto condizionato e vale la condizione $r_m \neq r_n$. Se viceversa si ha $r_m = r_n$ la computazione salta all'istruzione I_q.

I_1	I_1	I_1
I_2	I_2	I_2
.	.	.
$I_k = S(3)$	$I_k = C(3, 5, 9)$	$I_k = C(3, 5, 9)$
$\boldsymbol{I(k+1)}$	.	$\boldsymbol{I(k+1)}$
.	$\boldsymbol{I(9)}$	.
.	.	.
I_s	I_s	I_s

(a) Prossima istruzione nel caso di I_k istruzione aritmetica

(b) Prossima istruzione nel caso di $I_k = C(m, n, q)$ con $r_m = r_n$

(c) Prossima istruzione nel caso di $I_k = C(m, n, q)$ con $r_m \neq r_n$

Figura 8.3: I casi possibili per la prossima istruzione (in grassetto)

STOP della computazione - La computazione si ferma per due possibili motivi:

- È stata eseguita $I_k = I_s$ aritmetica, oppure $I_k = I_s = C(m, n, q)$ con $r_m \neq r_n$ (fig. 8.4a e 8.4b)
- È stata eseguita $I_k = C(m, n, q)$ con $r_m = r_n$ e $q > s$ (8.4c)

La prima delle due possibilità corrisponde all'interruzione della computazione per mancanza di altre istruzioni, in quanto è stata eseguita l'ultima istruzione del programma; nel secondo caso, invece, si ha un'uscita forzata poiché si è ottenuto il risultato richiesto e non è più necessario procedere oltre con la computazione. Ciò si ottiene richiedendo l'esecuzione di un'istruzione I_q il cui indice q non figura nell'elenco delle istruzioni del programma; ciò porta (convenzionalmente) a una terminazione della computazione. Nell'esempio di figura (8.4c) è stato posto $q = 99$; ciò fa intendere immediatamente che se vale la condizione $r_m = r_n$, allora ci sarà uno STOP nella computazione, poiché risulta evidente che gli esempi che faremo non avranno mai un numero di istruzioni superiore a 99.
Altro esito possible per la computazione è che essa entri in un ciclo e non giunga mai allo STOP. In questo caso si parla di computazione non terminante o *divergente*.

I_1	I_1	I_1
I_2	I_2	I_2
.	.	.
.	.	$I_k = C(3,5,99)$
.	.	.
$I_s = I_k = S(3)$	$I_s = I_k = C(3,5,9)$	I_{37}
(a) STOP della computazione con I_s=I_k aritmetica	(b) STOP della computazione con I_k=I_s=$C(m,n,q)$ e con $r_m \neq r_n$	(c) STOP della computazione con I_s=I_k=$C(m,n,q)$ e con r_m=r_n e q>s

Figura 8.4: I casi possibili per lo STOP della computazione

Configurazione iniziale - È la configurazione dalla quale si parte per effettuare la computazione. Se $a_1, a_2, ..., a_n$ sono i dati d'ingresso, per convenzione essi vengono posti all'inizio del nastro, lasciando a 0 tutte le altre (infinite) celle di memoria (fig. 8.5).

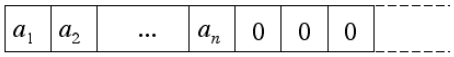


Figura 8.5: Configurazione iniziale del nastro caricato con i dai iniziali $a_1, a_2, ..., a_n$

Configurazione finale - È la configurazione che si ottiene alla fine della computazione. Per convenzione il valore b calcolato dalla computazione è il contenuto della prima cella. Ciò accade ovviamente nel solo caso in cui la computazione termini.

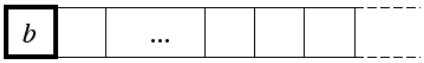
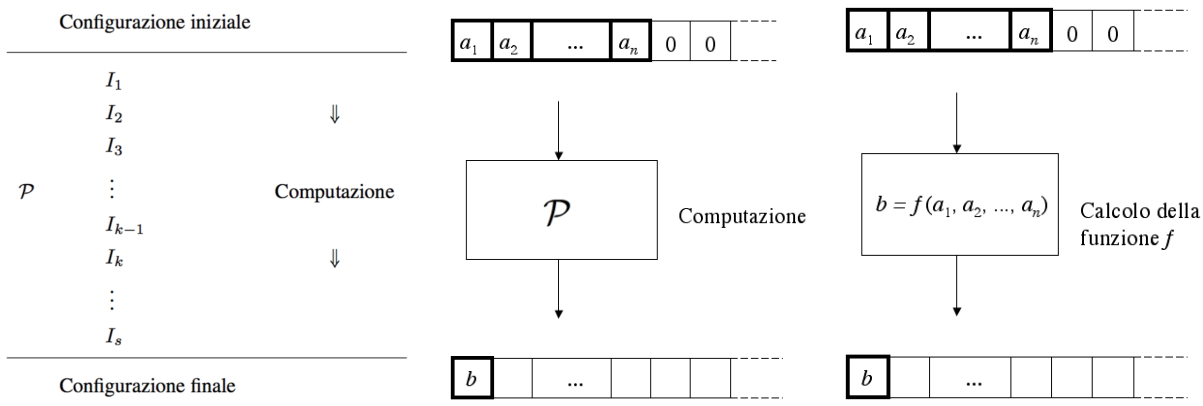


Figura 8.6: Configurazione finale del nastro: il contenuto della prima cella viene considerato il valore calcolato

Convergenza - Indichiamo con la notazione $\mathcal{P}(a_1, a_2, ..., a_n)$ la computazione del programma $\mathcal{P}$ a partire dalla configurazione iniziale $a_1, a_2, ..., a_n$. Se tale computazione termina diciamo che c'è stata *convergenza*, e scriviamo $\mathcal{P}(a_1, a_2, ..., a_n)\downarrow$. Se b è il contenuto della prima cella alla fine della computazione diciamo che la computazione è andata a convergenza su b e scriviamo $\mathcal{P}(a_1, a_2, ..., a_n)\downarrow b$. Se la computazione non termina diciamo che si è avuta una *divergenza* e scriviamo $\mathcal{P}(a_1, a_2, ..., a_n)\uparrow$

Analizziamo ora il significato della computazione del programma $\mathcal{P}$=$\{I_1, I_2, ..., I_s\}$ da un punto di vista più astratto. Sappiamo che a partire dalla configurazione iniziale del nastro si perviene alla configurazione finale, che corrisponde al *risultato* della computazione, cioè dell'esecuzione, passo passo, delle istruzioni del programma (fig.8.7a). La computazione è quindi un procedimento astratto il quale, a partire da una certa configurazione iniziale $x_1, x_2, ..., x_n$ restituisce in un tempo finito un certo valore y, leggibile sulla prima cella del nastro (fig.8.7b). Ma un procedimento che associ un numero intero y a un'n-pla $x_1, x_2, ..., x_n$ di interi corrisponde al calcolo della funzione $y = f(x_1, x_2, ..., x_n)$, che è una funzione $f : \mathbb{N}^n \rightarrow \mathbb{N}$ (fig.8.7c).
Invertendo i termini della questione possiamo affermare che, volendo *calcolare* la funzione y=$f(x_1, x_2, ..., x_n)$ possiamo far uso del programma $\mathcal{P}$. Se si vuole calcolare $f(a_1, a_2, ..., a_n)$, mettiamo i valori $a_1, a_2, ..., a_n$ come configurazione iniziale e facciamo partire la computazione; a seguito della convergenza dobbiamo trovare il valore



(a) La computazione del programma $\mathcal{P}$ corrisponde all'esecuzione delle sue istruzioni

(b) La computazione del programma $\mathcal{P}$ dal punto di vista astratto

(c) La computazione del programma $\mathcal{P}$ corrisponde al calcolo di una certa funzione f

Figura 8.7: Il significato della computazione RAM

b nella prima cella. Si osservi che la funzione potrebbe non essere definita per qualche n-pla d'ingresso, per esempio per la $a_1^*, a_2^*, ..., a_n^*$. È ovvio che in questa circostanza non possiamo far convergere la computazione, perché il valore letto nella prima cella alla fine della stessa verrebbe interpretato come *il* valore della funzione, che invece non è definita. L'unica possibilità che rimane è allora quella di far divergere la computazione nel caso in cui la funzione non sia definita.

Calcolo di una funzione tramite un programma Diciamo che il programma $\mathcal{P}$ RAM-calcola la funzione f se, $\forall a_1, a_2, ..., a_n, b$

$$\mathcal{P}(a_1, a_2, ..., a_n) \downarrow b \quad \Leftrightarrow \quad \begin{cases} a_1, a_2, ..., a_n \in \text{Dom}(f) \\ \\ b = f(a_1, a_2, ..., a_n) \end{cases}$$

il che implica che $\mathcal{P}(a_1, a_2, ..., a_n)\uparrow$ se e solo se $a_1, a_2, ..., a_n \notin \text{Dom}(f)$

Funzione calcolabile Una funzione f si dice *calcolabile* se esiste un programma $\mathcal{P}$ che la RAM-calcola.

Potrebbe sembrare una forzatura il fatto che, per esprimere il funzionamento e l'attività di un calcolatore, si debba ricorrere al linguaggio delle funzioni. In realtà ci si rende subito conto che questo è il linguaggio corretto per rappresentare qualunque attività del computer. Esso è infatti una macchina discreta che lavora in tempo discreto; immaginiamo allora di fare una "fotografia" dei suoi circuiti in un certo istante discreto t_0. Poiché il computer è costituito da milioni di transistors, ciascuno dei quali lavora secondo una logica binaria, in linea di principio potremmo fare una lista ordinata di tali transistors, scrivendo "0" o "1" a seconda che, all'istante t_0, il componente sia nello stato di piena conduzione o di interdizione. Il lungo elenco di zeri e uni corrisponde a un vettore binario, con un numero di coordinate uguale al numero di transistors, che descrive lo *stato* della macchina; indichiamo tale vettore con la notazione $a_1, a_2, ..., a_n$. Se passiamo ora all'istante successivo t_1, ci sarà stata un'evoluzione dello stato a seguito delle istruzioni del programma. Supponiamo per esempio che si stia spostando la freccia del *mouse* sullo schermo; potrebbe allora succedere che in t_0 un certo *pixel*, a 256 livelli di colore, fosse p.es. color bianco, caratterizzato dalla codifica 00001011, che corrisponde al numero 11 nella scala $0...255$. Durante lo spostamento, all'istante t_1 il *pixel* diventa color grigio scuro, e viene codificato dal vettore 11011010 che corrisponde al numero 218. Possiamo allora affermare che la descrizione del funzionamento del *pixel* in questione è basata sul calcolo

della funzione $f(a_1, a_2, ..., a_n) = 218$. È ovvio che, secondo questo approccio, serve una funzione per ciascun *pixel*, ma questo è un problema tutto quantitativo, legato alla circostanza che ci sono moltissimi *pixel*. Facciamo ora alcuni esempi di costruzione di programmi elementari che servono per spiegare il funzionamento del modello.

Esempi

Esempio 8.1. Si voglia scrivere un programma che calcola la funzione $f(x) = 0,\ \forall x$.

Si tratta di partire dalla configurazione iniziale $x, 0, 0, \ldots$ per giungere alla configurazione finale $0, \ldots$, qualunque sia il valore di x. Il "programma" è in questo caso costituito dalla sola istruzione $Z(1)$, che porta a zero la prima cella, qualunque sia il contenuto della stessa.

Esempio 8.2. Scriviamo un programma che calcola la funzione $f(x) = 3,\ \forall x$.

Partendo dalla configurazione $x, 0, 0, \ldots$ si deve giungere alla configurazione finale $3, \ldots$, qualunque sia il valore di x. Il programma è

1	$Z(1)$	(porta a zero la prima cella, qualunque sia x)
2	$S(1)$	(incrementa la prima cella)
3	$S(1)$	(incrementa la prima cella)
4	$S(1)$	(incrementa la prima cella)

Si noti che si può ottenere la stessa funzione incrementando per tre volte il contenuto di una qualunque cella diversa dalla prima (p.es. la seconda) e usando l'istruzione $T(m, n)$ per trasferire il risultato

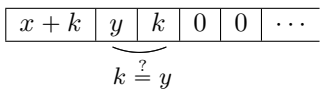
1	$S(2)$	(incrementa la seconda cella)
2	$S(2)$	(incrementa la seconda cella)
3	$S(2)$	(incrementa la seconda cella)
4	$T(2, 1)$	(copia nella prima cella il contenuto della seconda)

Esempio 8.3. Si voglia scrivere un programma che calcola la funzione $f(x) = x + 3,\ \forall x$.
Partendo dalla configurazione $x, 0, 0, \ldots$ si deve giungere alla configurazione finale $x + 3, \ldots$, qualunque sia il valore di x. Il programma è

1	$S(1)$	(incrementa la seconda cella)
2	$S(1)$	(incrementa la seconda cella)
3	$S(1)$	(incrementa la seconda cella)

Esempio 8.4. Si voglia scrivere un programma che calcola la funzione $f(x, y) = x + y,\ \forall (x, y)$.

Partendo dalla configurazione $x, y, 0, \ldots$ si deve giungere alla configurazione finale $x + y, \ldots$, qualunque siano i valori di x e y. Si noti che non possiamo inserire y comandi $S(1)$, poiché non conosciamo a priori il valore di y. Bisogna allora ricorrere alla seguente strategia: si incrementa progressivamente x, in modo che diventi $x + 1$, $x + 2$, $x + 3$, ..., $x + k$,... memorizzando nel contempo il valore corrente di k su una delle celle libere. Quando si giunge al valore k tale che $k = y$, allora nella prima cella c'è $x + y$. Si osservi che il nodo della computazione è il controllo $k \stackrel{?}{=} y$. Se $k \neq y$ dobbiamo continuare a incrementare k; se viceversa $k = y$ allora abbiamo concluso. Lo stato corrente della computaizone è allora

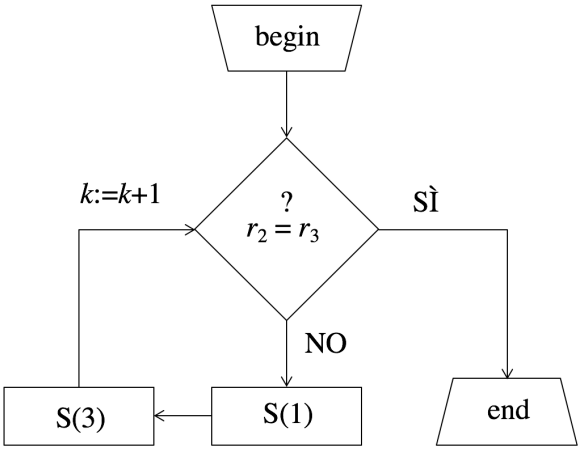


Per poter intraprendere due percorsi diversi di computazione a seconda che k sia o meno uguale a y, è necessario introdurre l'istruzione di salto condizionato $C(m, n, q)$, con m e n indirizzi di memoria dove sono memorizzati k e y. Si noti peraltro che il controllo deve essere svolto immediatamente, poiché potrebbe accadere che sia $y = 0$,

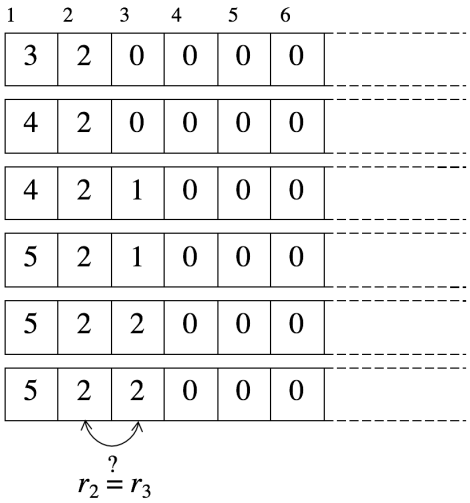
nel qual caso avremmo già terminato. Sarà dunque necessario iniziare con un controllo del tipo $C(2,3,99)$, che ci fa concludere la computazione se il contenuto della seconda cella coincide con quello della terza, cioè se $k = y$. Se viceversa $k \neq y$ si deve incrementare k, cioè $k := k + 1$[1] nella prima cella ($S(1)$) e nella terza cella ($S(3)$). A questo punto la nuova configurazione sarà $x + k + 1, y, k + 1, 0, 0, \ldots$, e dovremo rifare il controllo, per verificare se $k + 1 = y$ o meno. Senza inserire una nuova istruzione $C(2,3,99)$ nel programma, il controllo può essere fatto tornando alla prima istruzione $C(2,3,99)$ mediante un'istruzione di *salto incondizionato* del tipo $C(1,1,1)$; poiché è sempre vero che il contenuto della prima cella è uguale a sè stesso, si riaccede alla prima istruzione e si effettua nuovamente il controllo. In questo modo si crea un *ciclo*, dal quale si esce solo quando è verificata la condizione $r_2 = r_3$.

```
1   ┌→ C(2,3,99)   (controllo se k = y)
2   ¦   S(1)       (k := k + 1 nella prima cella)
3   ¦   S(3)       (k := k + 1 nella terza cella)
4   └-- C(1,1,1)   (salto incondizionato alla prima istruzione)
```

In figura 8.8a viene riportato il *diagramma di flusso* del programma, che mette in relazione le operazioni effettuate dallo stesso, mentre in figura 8.8b viene riportata la sequenza degli stati di memoria nella somma $3 + 2$.



(a) Diagramma a flusso del programma che calcola la funzione $f(x,y) = x + y$



(b) Successione degli stati per il calcolo della somma 3+2

Figura 8.8:

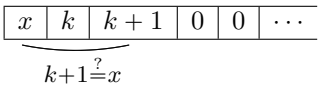
Esempio 8.5. Si voglia scrivere un programma che calcola la funzione $f(x) = x - 1$ sui numeri naturali.

Essa si indica col simbolo $\dot{-}$ e si definisce nel modo seguente

$$f(x) = x \dot{-} 1 = \begin{cases} x - 1 & x > 0 \\ 0 & x = 0 \end{cases} \quad \forall x$$

Per realizzare questa funzione dobbiamo costruire una differenza a partire dalle quattro istruzioni di base, che *non* contemplano alcun tipo di sottrazione. Una possibilità è quella di avere una configurazione corrente del tipo

[1] Il simbolo ":=" si legge "diventa".



in modo tale che quando $x = k + 1$, nella seconda cella si trova $k = x - 1$, che deve essere trasferito nella cella iniziale. Poiché la funzione vale 0 quando $x = 0$, la prima cosa da fare è la seguente verifica $x \stackrel{?}{=} 0$; se la risposta è sì allora abbiamo terminato e possiamo uscire, altrimenti si continua col programma. Si tratta allora di porre il $+1$ nella terza cella e fare la verifica $r_1 \stackrel{?}{=} r_3$; se la condizione è soddisfatta si esce, altrimenti $k := k + 1$ e inizia un ciclo.

```
1        C(1, 4, 99)     (controlla se x = 0 usando la quarta cella)
2        S(3)            (pone +1 nella terza cella)
3   ┌──► C(1, 3, 7)  ┆   (controlla se x = k + 1)
4   ┆    S(2)        ┆   (k := k + 1 nella seconda cella)
5   ┆    S(3)        ┆   (k := k + 1 nella terza cella)
6   └┄┄  C(1, 1, 3)     (salto incondizionato alla terza istruzione)
7        T(2, 1)  ◄┄┄┘   (trasferisce il risultato x − 1 nella prima cella)
```

In figura 8.9 viene riportato il *diagramma a flusso* del programma, che mette in relazione le operazioni effettuate dallo stesso.

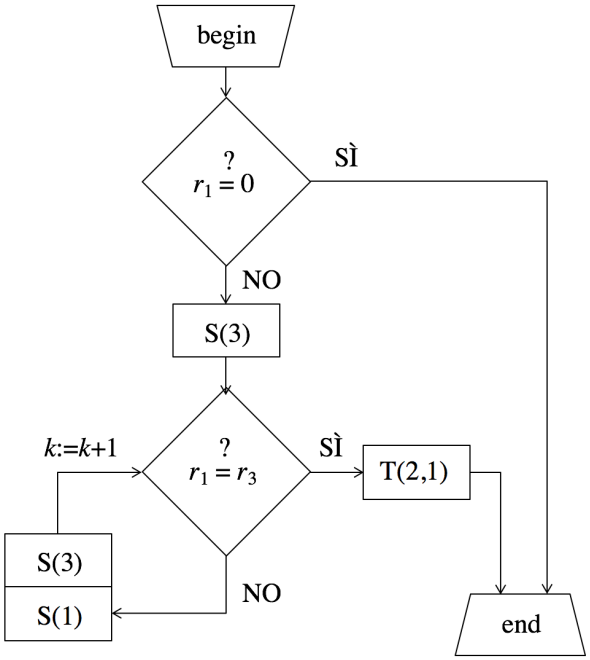


Figura 8.9: Diagramma a flusso del programma che calcola la funzione $f(x) = x \dot{-} 1$

Esempio 8.6. Si voglia scrivere un programma che calcola la funzione $f(x) = x - y$ sui numeri naturali, che si definisce nel modo seguente

$$f(x) = x \dot{-} y = \begin{cases} x - y & x \geqslant y \\ \text{indef} & x < y \end{cases} \quad \forall (x, y)$$

Anche in questo caso si tratta di costruire una differenza a partire dalle quattro istruzioni di base, che *non* contemplano alcun tipo di sottrazione. Quando la funzione è definita si ha sempre $x \geqslant y$, e la differenza $x - y$

corrisponde al numero di volte in cui bisogna incrementare y per arrivare a x. La soluzione è allora quella di avere una configurazione corrente del tipo

x	$y+k$	k	0	0	$\cdots$

$y+k \stackrel{?}{=} x$

in modo tale che, quando $x = y + k$, nella terza cella si trova $k = x - y$, che deve essere poi trasferito nella cella iniziale. Anche in questa caso bisogna iniziare con un controllo, poiché potrebbe succedere che sia $x = y$, nel qual caso sarebbe già tutto finito.

```
1   C(1,2,5)   (controlla se x = y + k)
2   S(2)       (k := k + 1 nella seconda cella)
3   S(3)       (k := k + 1 nella terza cella)
4   C(1,1,1)   (salto incondizionato alla prima istruzione)
5   T(3,1)     (trasferisce il risultato x − y nella prima cella)
```

In figura 8.10 viene riportato il *diagramma a flusso* del programma, che mette in relazione le operazioni effettuate dallo stesso. Osserviamo che se $x < y$, sarà sempre vero che $x < y+k$, e quindi il confronto tra r_1 e r_2 della prima

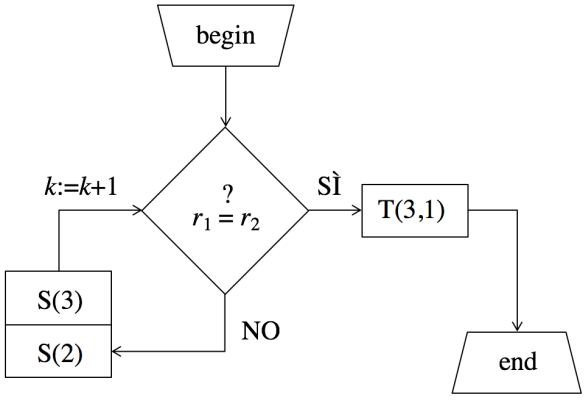


Figura 8.10: Diagramma a flusso del programma che calcola la funzione $f(x) = x \dot{-} 1$

istruzione $C(1, 2, 5)$ darà sempre esito negativo; non ci sarà quindi la possibilità di uscire dal ciclo per terminare con l'istruzione $T(3, 1)$ e la macchina ciclerà all'infinito. Ciò è coerente col fatto che per $x < y$ la funzione risulta non definita.

Esempio 8.7. Scriviamo un programma che calcola la funzione

$$f(x) = \begin{cases} x/2 & \text{se } x \text{ pari} \\ \text{indef} & \text{se } x \text{ dispari} \end{cases} \quad \forall x$$

La configurazione corrente che risolve il problema è $x, k, 2k, 0, 0, \ldots$, in modo tale che quando $x = 2k$ nella seconda cella si ha $k = x/2$. Ogniqualvolta incrementiamo k di un'unità nella seconda cella, dobbiamo incrementare di due unità la terza

```
1   C(1,3,6)   (controlla se x = 2k)
2   S(2)       (k := k + 1 nella seconda cella)
3   S(3)       (k := k + 1 nella terza cella)
4   S(3)       (k := k + 2 nella terza cella)
5   C(1,1,1)   (salto incondizionato alla prima istruzione)
6   T(2,1)     (trasferisce il risultato k = x/2 nella prima cella)
```

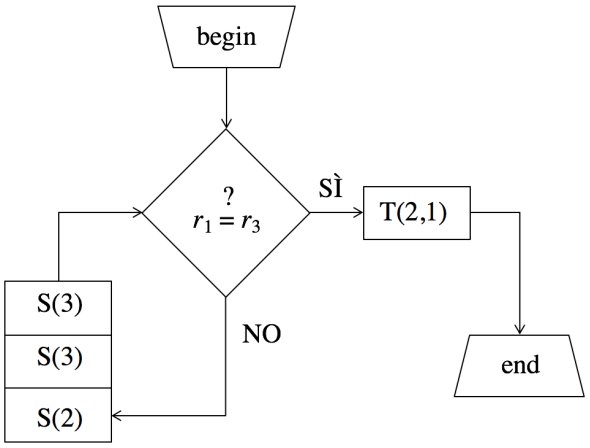


Figura 8.11: Diagramma a flusso del programma che calcola la funzione $f(x) = x/2$

Esempio 8.8. Scriviamo un programma che calcola le seguenti funzioni

$$f(x) = \begin{cases} 1 & \text{se } x = 0 \\ 0 & \text{se } x \neq 0 \end{cases} \quad \forall x$$

1	$C(1,2,4)$
2	$Z(1)$
3	$C(1,1,99)$
4	$S(1)$

$$f(x) = \begin{cases} 0 & \text{se } x = 0 \\ 1 & \text{se } x \neq 0 \end{cases} \quad \forall x$$

1	$C(1,2,99)$
2	$Z(1)$
3	$S(1)$

Esempio 8.9. Scriviamo un programma che calcola le seguenti funzioni

$$f(x,y) = \begin{cases} 1 & \text{se } x = y \\ 0 & \text{se } x \neq y \end{cases} \quad \forall (x,y)$$

1	$C(1,2,4)$
2	$Z(1)$
3	$C(1,1,99)$
4	$Z(1)$
5	$S(1)$

$$f(x,y) = \begin{cases} 0 & \text{se } x = y \\ 1 & \text{se } x \neq y \end{cases} \quad \forall (x,y)$$

1	$C(1,2,5)$
2	$Z(1)$
3	$S(1)$
4	$C(1,1,99)$
5	$Z(1)$

Selezione e iterazione col linguaggio del modello RAM

Abbiamo più volte sottolineato il fatto che l'approccio procedurale-algoritmico fa riferimento al concetto di programma $\mathcal{P}$, costituito da un elenco di istruzioni $I_1, I_2, \ldots, I_s$, che devono essere eseguite in sequenza. Questo è l'approccio seguito nell'ambito della programmazione imperativa, dominante rispetto ad altri paradigmi. Ogni istruzione corrisponde a un "comando" che viene impartito alla macchina, e che prevede l'esecuzione di un certo lavoro. Nella sezione 7.1.1 abbiamo fissato le caratteristiche del *linguaggio macchina* che dà corpo a tali istruzioni; abbiamo visto che si tratta di istruzioni molto "povere", nel senso che dovendo essere direttamente eseguibili dal processore non possono prevedere elaborazioni troppo complesse, ma si limitano a operazioni di base del tipo "somma il contenuto di due celle di memoria", oppure "copia il contenuto di un cella in un'altra", "azzera il contenuto di una cella", ecc. Le istruzioni del linguaggio RAM sono proprio di questo tipo, e in tal senso costituiscono

un modello molto realistico del funzionamento del calcolatore a livello *hardware*.
Tuttavia, quando si deve realizzare un programma non si lavora mai a livello di linguaggio macchina, poiché sarebbe inutilmente faticoso, inefficiente e frustrante. Si preferisce invece operare a un livello logico superiore, usando linguaggi come C, C++, Java, Fortran, Pascal, ecc. per i quali le istruzioni elementari consentono di effettuare operazioni logiche più complesse di quanto si possa realizzare in linguaggio macchina, molto vicine alla logica che guida il ragionamento umano. In questo modo c'è anche il vantaggio che i programmi in tali linguaggi sono indipendenti dall'architettura della macchina. Ricordiamo inoltre che la traduzione tra un linguaggio ad *alto livello* di questo tipo e il linguaggio macchina, viene garantita da un programma che si chiama *compilatore*, e che andrà adattato alle diverse piattaforme architetturali in uso.
Se facciamo riferimento al paradigma più usato di programmazione imperativa, denominato *programmazione strutturata*, possiamo affermare che un programma è solitamente costruito nel seguente modo:

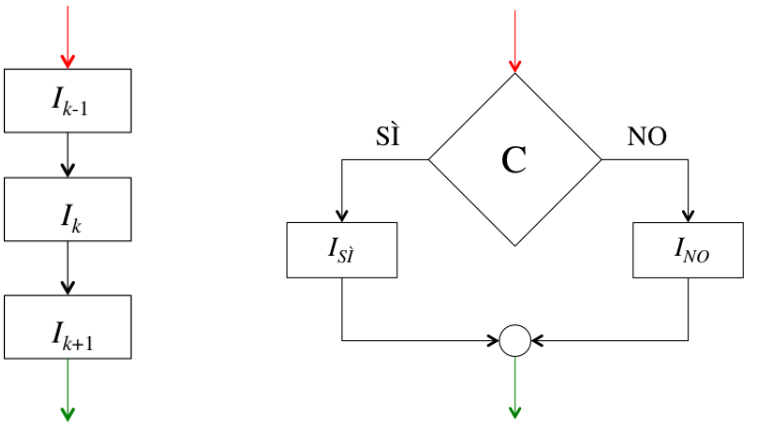
- *una parte dichiarativa*, in cui si dichiarano tutte le variabili del programma e il loro tipo (p.es. variabile intera, variabile carattere, ecc);

- *una parte che descrive l'algoritmo* risolutivo utilizzato, basato sulle istruzioni del linguaggio;
  a loro volta le istruzioni si dividono in:

  - istruzioni di lettura e scrittura (scrittura a video, scrittura su disco, lettura da tastiera, ...);
  - istruzioni di assegnamento (del valore a una variabile);
  - istruzioni logiche di controllo (frasi if, while, for, repeat, case, ...).

In un linguaggio strutturato, quale ad esempio il Pascal, ci sono sostanzialmente tre tipi di *strutture logiche di controllo* del programma; esse sono rispettivamente la *Sequenza*, la *Selezione* e l'*Iterazione*. Analizziamole nei dettagli:

Nella **Sequenza** (Fig. 8.12a) le istruzioni sono semplicemente poste in sequenza, una dopo l'altra. Il punto d'ingresso è evidenziato da una freccia rossa, quello di uscita da una freccia verde.



(a) Sequenza di istruzioni

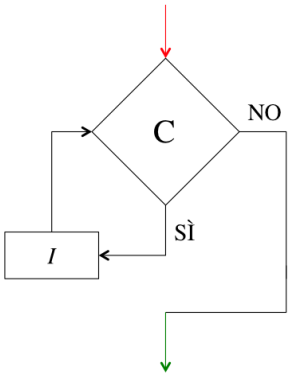
(b) Selezione effettuata sulla base della validità della condizione C; corrisponde a una frase Pascal **if** C **then** $I_{SÌ}$ **else** I_{NO}

Figura 8.12: Strutture logiche di controllo del tipo *sequenza* e *selezione*

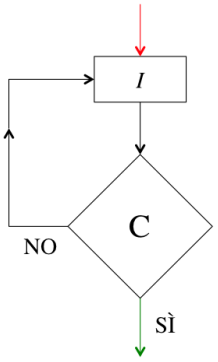
Nella **Selezione** (Fig. 8.12b) si procede a una verifica della validità della condizione C, seguendo due strade diverse a seconda che C sia o meno vera; se C è vera (SÌ) si procede con l'esecuzione dell'istruzione $I_{SÌ}$, altrimenti (NO) si esegue l'istruzione I_{NO}. Nel linguaggio Pascal la selezione corrisponde alla frase **if** C **then** $I_{SÌ}$ **else** I_{NO}

Nell'**Iterazione** (Fig. 8.13) si attiva un *ciclo*, controllato dalla realizzazione di una certa condizione prefissata. Il

controllo può avvenire in due modi diversi: nel primo la validità della condizione C viene verificata subito; se essa è soddisfatta si esegue I altrimenti si esce. In questo caso si rimane nel ciclo e si continua a eseguire I finché la condizione vale; ciò corrisponde a una frase Pascal del tipo **while** C **do** I (si veda la figura 8.13a). Nel secondo caso viene eseguita subito l'istruzione I e solo successivamente si attua la verifica della condizione C; se essa non è soddisfatta si ri-esegue I altrimenti si esce; ciò corrisponde a una frase Pascal del tipo **repeat** I **until** C (si veda la figura 8.13b). Si osservi che in questo secondo caso l'istruzione I viene eseguita almeno una volta.



(a) Iterazione del tipo **while** C **do** I



(b) Iterazione del tipo **repeat** I **until** C

Figura 8.13: Due tipi diversi di iterazione

Immaginiamo ora che il linguaggio macchina del calcolatore sia basato sul linguaggio del modello RAM, e che si voglia "compilare", cioè tradurre una frase complessa del linguaggio Pascal in una sequenza di istruzioni RAM. Il punto nodale è chiaramente la costruzione delle strutture logiche di controllo, perché per il resto si tratta solo di assegnazioni o di operazioni di *input/output* dei dati. Si tratta quindi di costruire un programma RAM per ciascuna delle tre frasi **if-then-else**, **while-do** e **repeat-until**, in modo da poter dare le istruzioni corrette a livello macchina quando una di queste frasi venga evocata su un programma di un linguaggio ad alto livello. Cominciamo con la prima.

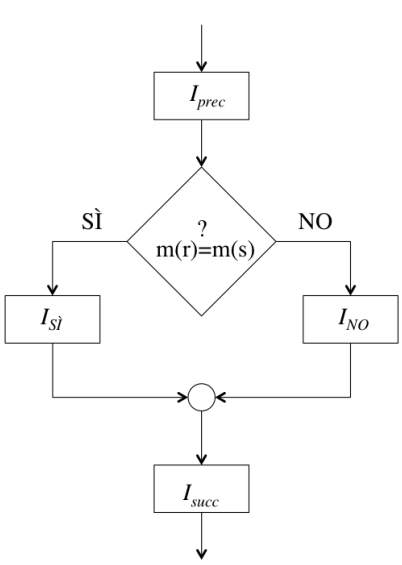
Frase di selezione **if** C **then** $I_{SÌ}$ **else** I_{NO}

Immaginiamo che la frase sia preceduta da una generica istruzione RAM che chiamiamo I_{prec} e seguita da (da una generica istruzione RAM che chiamiamo) I_{succ}. Se abbiamo già eseguito l'istruzione I_{prec} dobbiamo ora scegliere uno dei due percorsi, che portano a una delle due istruzioni $I_{SÌ}$ o I_{NO}, a seconda che la condizione C sia o meno vera. Per effettuare un tale tipo di verifica il linguaggio RAM ci offre l'istruzione $C(m, n, q)$; la condizione C di figura 8.12b deve essere dunque ricondotta a una verifica del tipo $r_m \stackrel{?}{=} r_n$. Se la risposta è sì si va a sinistra e si esegue $I_{SÌ}$; se la risposta è no si va a destra e si esegue I_{NO}. Sotto riportiamo le righe del codice associato.

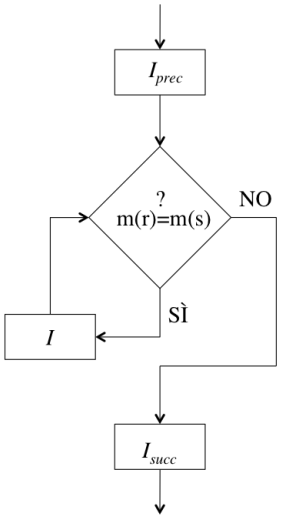
$k-1$	I_{prec}	(esegue l'istruzione precedente)
k	$C(m, n, k+3)$	(controlla se $r_m = r_n$)
$k+1$	I_{NO}	(se $r_m \neq r_n$ si esegue I_{NO})
$k+2$	$C(1, 1, k+4)$	(passa a I_{succ}, senza eseguire anche $I_{SÌ}$)
$k+3$	I_{SI}	(se $r_m = r_n$ si esegue $I_{SÌ}$)
$k+4$	I_{succ}	(esegue l'istruzione successiva)

Frase di iterazione **while** C **do** I

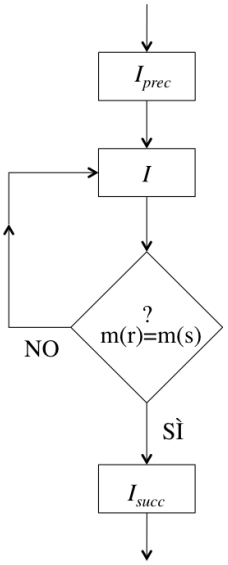
Eseguita I_{prec} si deve verificare la validità della condizione $r_m \stackrel{?}{=} r_n$. Se la condizione è soddifatta si innesca un ciclo che prevede l'esecuzione dell'istruzione I; tale istruzione non può tuttavia seguire immediatamente l'istru-



(a) Diagramma di flusso della frase **if-then-else**



(b) Diagramma di flusso della frase **while-do**



(c) Diagramma di flusso della frase **repeat-until**

Figura 8.14: Diagrammi di flusso delle frasi principali di selezione e di ciclo

zione di controllo, poiché l'istruzione immediatamente successiva a $C(m, n, .)$ è quella che viene attivata nel caso in cui la condizione *non* fosse stata soddisfatta; bisogna dunque fare un salto incondizionato $C(1, 1, k + 2)$ alla riga $k + 2$. Bisogna poi rifare la verifica $r_m = r_n?$, e per questo si attiva un salto incondizionato $C(1, 1, k)$, che ci riporta al passo k. Finché la condizione $r_m = r_n$ continua a essere soddisfatta si rimane nel ciclo. Nel caso invece la condizione non sia più soddisfatta si deve uscire dal ciclo, saltando direttamente all'istruzione successiva, di riga $k + 4$, con un salto incondizionato $C(1, 1, k + 4)$.

$k-1$	I_{prec}	(esegue l'istruzione precedente)
k	$C(m, n, k+2)$	(controlla se $r_m = r_n$)
$k+1$	$C(1, 1, k+4)$	(se $r_m \neq r_n$ si deve andare a I_{succ})
$k+2$	I	(se $r_m = r_n$ si esegue I)
$k+3$	$C(1, 1, k)$	(salto incondizionato a I_k che chiude un ciclo)
$k+4$	I_{succ}	(esegue l'istruzione successiva)

Frase di iterazione **repeat** I **until** C

Subito dopo aver eseguito I_{prec} si esegue immediatamente anche la I. Poi si attua la verifica della condizione $r_m = r_n?$. Se la condizione non è soddisfatta si apre un ciclo con un'istruzione di salto incondizionato $C(1, 1, k)$, che ci riporta alla riga k; finché $r_m \neq r_n$ si rimane nel ciclo eseguendo più volte la I. Quando la C diventa vera si esce dal ciclo, andando all'istruzione successiva.

$k-1$	I_{prec}	(esegue l'istruzione precedente)
k	I	(esegue I)
$k+1$	$C(m, n, k+3)$	(se $r_m = r_n$ si va a I_{succ})
$k+2$	$C(1, 1, k)$	(se $r_m \neq r_n$ si apre un ciclo ritornando a I_k)
$k+3$	I_{succ}	(esegue l'istruzione successiva)

La costruzione dei programmi che sono il risultato della traduzione in linguaggio RAM delle tre frasi **if-then-else**, **while-do** e **repeat-until** ci dà la possibilità di creare il raccordo concettuale che esiste tra la scrittura di un programma ad alto livello e l'esecuzione delle corrispondenti istruzioni a seguito della compilazione nel linguaggio

macchina. Queste ultime, nel modello di calcolatore sotteso dal linguaggio RAM, occuperebbero le celle della memoria RAM illustrata sulla parte destra della figura 7.7, che descirive il ciclo *fetch-decode-execute* per l'esecuzione di un'istruzione di linguaggio macchina.