# Fondamenti di informatica: calcolabilità e complessità

*Pierpaolo Degano*
Dipartimento di Informatica, Università di Pisa
Largo Bruno Pontecorvo, 3, I-56127 Pisa, Italia
degano@di.unipi.it

23 ottobre 2025

**Sommario**

AVVERTENZA: queste note raccolgono appunti per l'insegnante in forma *molto preliminare*; di conseguenza sono scritte in un italiano assai approssimativo e spesso troppo schematico; contengono moltissimi errori di notazione e di stumpa; spesso fanno riferimento a parti o disegni non ancora presenti; hanno commenti che sono comprensibili a stento da chi li ha scritti; non hanno alcuna unità di presentazione né alcuna ambizione a fornire altro che una sommaria guida agli argomenti trattati a lezione.

Desidero ringraziare Stefania Pellegrini e Davide Marchignoli senza il cui aiuto non sarei mai riuscito a dare una forma decente, e men che meno stampabile, a questi appunti. Inoltre ringrazio gli studenti che mi hanno segnalato errori di ogni tipo, anche per il piacere che mi ha dato conoscerli e apprezzarli; un grazie speciale all'inesausto "Anagrammatore" che dal mio nome ha tratto *Odino è al Po: prega!*, stigmatizzando in me sia Ûdin, mia materna città che Attila, forse in omaggio a tale barbaro dio, dicon fondasse, sia il livido terrore che mi piacerebbe i malcapitati, pallidi, almeno fingessero all'esame. Inoltre ho molto apprezzato la fantasia di Alessandra che enumerando le lettere dell'alfabeto italiano e sommando i numeri corrispondenti a quelle del mio cognome ha ovviamente trovato la galattica risposta: 42; né posso omettere di menzionare l'autrice della foto qui sotto — alias la strega Cespuglio — e il soggetto della stessa e i loro compagni, splendidi per simpatia e bravura.

Un particolare ringraziamento va a Clelia De Felice per aver letto e commentato queste note con estrema accuratezza, per i suoi profondi suggerimenti, preziosissimi anche quando non accolti.

# Indice

**Introduzione** . . . . . 3

**1 Elementi di Calcolabilità** **7**
1.1 Idea intuitiva di algoritmo . . . . . 8
1.2 Macchina di Turing . . . . . 9
1.3 Linguaggi *FOR* e *WHILE* . . . . . 16
1.4 Problemi e Funzioni . . . . . 21
1.5 Due approcci alla calcolabilità . . . . . 27
1.6 Funzioni ricorsive primitive . . . . . 29
1.7 Diagonalizzazione . . . . . 39
1.8 Funzioni ricorsive generali . . . . . 41
1.9 Alcuni risultati classici . . . . . 44
1.10 Problemi insolubili e riducibilità . . . . . 58

**2 Elementi di Teoria dei Linguaggi Formali** **73**
2.1 Problemi come linguaggi . . . . . 74
2.2 Preliminari e notazione . . . . . 76
2.3 Grammatiche generative . . . . . 78
2.3.1 Gerarchia di Chomsky . . . . . 80
2.3.2 Linguaggi formali e di programmazione . . . . . 83
2.3.3 Derivazioni canoniche, ambiguità e alberi di derivazione e di sintassi astratta . . . . . 85
2.4 Linguaggi regolari . . . . . 90
2.4.1 Automi a stati finiti non-deterministici . . . . . 96
2.4.2 Espressioni Regolari. . . . . 100
2.4.3 Alcune proprietà dei linguaggi regolari . . . . . 101
2.5 Linguaggi liberi . . . . . 104
2.5.1 Automi a pila . . . . . 104
2.5.2 Alcune proprietà dei linguaggi liberi . . . . . 109
2.5.3 Linguaggi deterministici . . . . . 111
2.5.4 Parsing top-down e linguaggi LL . . . . . 116
2.5.5 Parsing bottom-up e linguaggi LR . . . . . 120

**3 Elementi di Complessità** **127**
3.1 Una teoria quantitativa degli algoritmi . . . . . 128
3.2 Misure di complessità deterministiche . . . . . 133
3.2.1 Macchine di Turing con k-nastri . . . . . 133
3.2.2 Complessità in tempo deterministico . . . . . 135
3.2.3 Macchine di Turing I/O . . . . . 146
3.2.4 Complessità in spazio deterministico . . . . . 147
3.3 Misure di complessità non deterministiche . . . . . 150
3.3.1 Macchine di Turing non deterministiche . . . . . 152
3.3.2 Complessità in tempo e in spazio non deterministici . . . . . 155
3.4 Funzioni di misura, un po' di gerarchia e due assiomi . . . . . 161
3.5 $\mathcal{P}$ e $\mathcal{NP}$ . . . . . 167
3.5.1 Problemi trattabili e problemi intrattabili . . . . . 168
3.5.2 Alcuni problemi interessanti e riduzioni efficienti tra essi 172
3.5.3 Problemi completi per $\mathcal{P}$ ed $\mathcal{NP}$ . . . . . 182

# Introduzione

Tutta da fare!!!

*Studia prima la scienza, e poi seguita la pratica, nata da essa scienza.*
*Quelli che s'innamoran di pratica senza scienza*
*son come 'l nocchier ch'entra in navilio senza timone e bussola,*
*che mai ha certezza dove si vada* — Leonardo da Vinci

Però la scienza che non genera pratica appassisce subito ...

*I wish I had more time to code, i.e. to prove*

**Scopo:** stabilire le potenzialità e i limiti del calcolo, formalizzandone in modo esatto (cioè *matematico*) l'intuizione — il rigore empirico non basta. Tutto ciò senza dimenticare che la nostra può non essere *l'unica* intuizione; se però molte "intuizioni" diverse convergono — si riducono l'una all'altra — allora abbiamo trovato una "buona" intuizione. Notare che viene "modellizzato" un concetto fisico e che quindi non si riesce, né si riuscirà a dimostrare l'eguaglianza del "calcolo" fisico con la sua formalizzazione matematica. Prego notare che seguiamo l'approccio galileano: si descrive con equazioni matematiche la realtà, sia pure artificiale. — nota: la macchina di Turing *precede* il primo calcolatore!

**Storia:** vedi:
- cabbala (dei nomi, "per scoprire Dio") — Abulafia (1240 Zaragoza)
- Ars Magna (per rappresentare e costruire verità, ovvero comunicare in una lingua "perfetta") — Ramon Lull, o Lulli 1235-1316, catalano/arabo, visita Leonardo Fibonacci a Pisa
- Leibnitz (1646-1716): a partire dai "primitivi" dedurre verità mediante regole logiche di combinazione, *il pensiero cieco*, ovvero "apparato di *calcolo* per trovare pensieri" (notevole il suo interesse per il calcolo binario)

In mezzo al fiorire di tentativi per costriure macchine che approssimavano o risolvevano equazioni di vario tipo, per esempio per calcolare integrali, uno dei primo approcci che conducono alla teoria "odierna" nasce alla Conferenza di Parigi del 1900 con Hilbert,[1] che dice:

[1]Hilbert suggerí 23 problemi da affrontare che son risultati essere di fondamentale importanza per la matematica e l'informatica e che al tempo erano irrisolti (ad oggi tre sono

> "ogni *problema matematico* deve necessariamente avere una caratterizzazione esatta, sia sotto forma di una soluzione esatta, sia mediante la dimostrazione dell'impossibilità della sua soluzione e di tutti i tentativi per raggiungerla."

C'erano dei problemi per i quali a lungo non si conosceva la soluzione, alcuni dei quali avevano da poco ricevuto una soluzione negativa, per esempio quello relativo alla "dimostrazione" dell'indipendenza dell'assioma delle parallele [Lobacevskij (Bolyai + Gauss) e Riemann (e Saccheri, intorno al 1700, che diceva che la geometria iperbolica "distrugge sé stessa" — per un punto esterno a una retta ne passano almeno due che le sono parallele — e quella ellittica "ripugna la natura della linea retta" — due rette qualsiasi hanno almeno un punto a comune)] o della quadratura del cerchio [Lindemann, 1882].
Hilbert riscrive l'Entscheidungsproblem nel 1928, qualificandolo come "problema basilare della logica":

> "l'Entscheidungsproblem è risolto se si conosce una *procedura* (!) che permette di decidere, con un numero *finito* di operazioni, la validità o la soddisfacibilità di una data espressione logica."

Primo fiero colpo all'Entscheidungsproblem è il teorema di incompletezza (Gödel, 1931): per ogni formalizzazione coerente (e sufficientemente generale) dei numeri naturali, esistono proposizioni vere che non si possono dimostrare. Quindi l'affermazione *non ignorabimus* viene dimostrata errata.

Successivamente, Church e Turing (1936) dimostrano indipendentemente che l'Entscheidungsproblem non ha soluzione positiva e *formalizzano la nozione di procedura* (via $\lambda$-calcolo e Macchine di Turing).

---

ancora aperti, nove risolti positivamente o negativamente, dieci parzialmente risolti, mentre uno riguarda il calcolo delle variazioni che è un ramo dell'analisi in continuo sviluppo). Oltre il citato 'Entscheidungsproblem, i problemi che riguardano in particolare la nostra disciplina sono i seguenti e hanno ricevuto tutti una risposta negativa [Gödel; Turing; Cohen, Scott e Solovay; Bowman–Robinson; Matjiasevic]:

- Ammesso che il ragionamento matematico sia finito, dimostrare con metodi finiti che la matematica è coerente.
- Dimostrare che la teoria degli insiemi è coerente.
- Dimostrare l'ipotesi del continuo.
- *Costruire* un algoritmo che decide quali equazioni diofantee hanno soluzioni.

Anche Kleene e Post nel 1936 e poi Markov, Chomsky ecc. propongono formalizzazioni diverse; la cosa importante è che *tutte* definiscono la *stessa* nozione di *problema* (cioè le formalizzazioni *si riducono* l'una all'altra) ed esibiscono un problema che è *insolubile*.

Come si traduce questo in termini informatici? Che riflesso ha sull'uso delle macchine, sulla definizione di *algoritmi*, scrittura di *programmi*, progettazione e costruzione di *linguaggi* di programmazione e degli strumenti di supporto (compilatori, interpreti, ...) e sistemi di calcolo?

La *Teoria della Calcolabilità* che introdurremo brevemente nella prima parte del corso studia che cosa può essere calcolato da un computer senza limitazioni di risorse spazio/temporali/energetiche o altro; ovvero:

i) quali sono i problemi *solubili* mediante una *procedura effettiva* (in un qualunque linguaggio, con una qualunque macchina)?

ii) esistono problemi *insolubili*? sono interessanti o puramente artificiali?

iii) si possono raggruppare i problemi in classi?

iv) quali sono le proprietà (delle classi) dei problemi solubili?

v) quali sono le relazioni tra (le classi de)i problemi insolubili?

Ci capiterà di vedere il meccanismo di calcolo come un automa che accetta gli elementi di un particolare insieme. Se questo è composto da stringhe su un alfabeto, tale insieme prende il nome di *linguaggio (formale)*. Vista l'importanza che i linguaggi formali hanno all'interno della nostra disciplina, dedicheremo la seconda parte del corso a esaminare piú in dettaglio la loro struttura, la loro capacità espressiva, quali problemi in tale area sono decidibili e quali efficientemente, con speciale attenzione agli aspetti che piú da vicino riguardano i linguaggi di programmazione, in particolare la costruzione di parti di compilatori e interpreti e di altri strumenti di uso comune. Vedremo che alcuni risultati teorici hanno avuto un grandissimo impatto sullo sviluppo del sofware, permettendo la generazione automatica e poco costosa di quelli strumenti di sostegno alla programmazione menzionati sopra.

Infine, nella terza parte del corso, ci restringeremo ai problemi solubili. Noteremo che vi è una gerarchia di classi (funzioni primitive ricorsive [Hilbert, Ackermann], sottoclassi [funzioni semplici, di Kalmar, gerarchia di Grzegorcicz], ...) che stanno alla base della *Teoria della Complessità*, il cui scopo è quantificare le risorse, principalmente in tempo o spazio o energia, necessarie a risolvere un problema dato, in funzione della sua taglia. Ovvero:

i) quali sono i problemi che sono solubili/insolubili *entro* un dato limite spazio/temporale? e viceversa: quali sono le risorse *minime* per risolvere una data classe di problemi? Esiste un legame tra queste classi e analoghe classi di linguaggi e di programmazione e loro costrutti?

ii) quali sono e come si misura il consumo delle risorse?

iii) ci sono dei limiti entro cui problemi noti *non* possono essere risolti? (e in questo caso ci possiamo accontentare di una soluzione approssimata? e che vuol dire approssimata?).

iv) ci sono problemi che richiedono più risorse di altri? e quali sono le caratteristiche che li rendono cosí "difficili"? Inoltre, dato un problema, esiste l'algoritmo ottimo per risolverlo? se sí, qual è la sua complessità?

v) quali sono le classi e le proprietà dei problemi solubili entro un dato limite? ce n'è di paradigmatiche? e innalzando tale limite, si allarga la classe dei problemi solubili?

Preliminare a questo studio sono le definizioni di:

- procedure effettive (Macchine di Turing, funzioni ricorsive generali, programmi `while`, ...) = *algoritmi*;
- problemi solubili e problemi insolubili.

## Nota a margine

Nonostante la presentazione dei risultati principali della teoria della calcolabilità e della complessità non possa che essere rigorosmente formale e costituisca quindi il supporto indispensabile alla realizzazione di ogni manufatto informatico, dalla progettazione di sistemi hardware e software, alle applicazioni di ogni natura, cercheremo di evidenziare come le definizioni e i teoremi che andremo a presentare si riverberano e si inverano nella prassi e nelle tecniche tipiche della nostra disciplina.

Naturalmente, ciò che andremo a vedere non ha certo l'ampiezza che lo studio degli algoritmi, sia qualitativo che quantitativo, e non solo perché ne caratterizzeremo solo una classe ristretta, ma anche perché trascuriamo alcuni loro aspetti affatto secondari, per esempio quelli non funzionali, quali quelli relativi alla loro robustezza e resilienza o alla loro capacità di resistere ad avversari che tentino di usarli per scopi malevoli.

# Capitolo 1

# Elementi di Calcolabilità

## 1.1 Idea intuitiva di algoritmo

Ci sono moltisimi formalismi che sono stati proposti per esprimere algoritmi, tra cui ricordiamo Macchine di Turing, funzioni ricorsive, $\lambda$-calcolo, Random Access Machine, algoritmi di Markov, grammatiche generali (v. la seconda parte), sistemi di Post e linguaggi di programmazione. In ciascuno di questi gli algoritmi devono soddisfare i seguenti requisiti:

i) Un algoritmo è costituito da un insieme *finito* di istruzioni;

ii) Le istruzioni possibili sono in numero *finito* e hanno un effetto *limitato* su dati *discreti*, esprimibili in maniera *finita*;

iii) Una computazione è eseguita per *passi discreti* (singoli), senza ricorrere a sistemi analogici o metodi continui, ciascuno dei quali impiega un tempo *finito*;

iv) Ogni passo dipende *solo* dai precedenti e da una porzione *finita* dei dati, in modo *deterministico* (cioè senza essere soggetti ad alcuna distribuzione probabilistica non banale);

v) *non c'è limite* al *numero di passi* necessari all'esecuzione di un algoritmo, nè alla *memoria* richiesta per contenere i dati (finiti) iniziali, intermedi ed eventualmente finali (si noti che il numero dei passi di calcolo è finito solo quando non vi sia alcuna istruzione dell'algoritmo che si possa eseguire, sia perché abbiamo trovato il risultato e raggiunto uno stato finale, sia perché ci troviamo in uno stato di stallo).

Sotto queste ipotesi, tutte le formulazioni fin ad ora sviluppate sono equivalenti e si *postula* che lo saranno anche tutte le future. [1]

**Nota storica**. Il termine *algoritmo* è documentato in lingua italiana fino dal 1300: nella *Nuova Cronica* Giovanni Villani scrive che a Firenze un migliaio di ragazzi "stanno ad apprendere l'algebra e l'*algorisimo*" dove però la parola significa il calcolo con le cifre arabe — a Venezia quelli che imparavano a calcolare erano molti di piú e a Pisa avevan iniziato ben prima, grazie a Leonardo Fibonacci che ne introdusse l'uso appreso a Bugia.

[1] Un'eccezione è costituita dalle macchine concorrenti/interattive in cui i dati di ingresso variano nel tempo; inoltre vi sono formalismi che tengono conto di quantità continue quali gli algoritmi probabilistici o stocastici, usati per esempio nella descrizione di sistemi biologici o di sistemi ibridi, anche se tali quantità sono poi approssimate a valori discreti, ricadendo cosí nel caso che consideriamo qui; altre eccezioni sono gli algoritmi nondeterministici, ma per ogni algoritmo di quest'ultimo tipo vi è un algoritmo deterministico del tutto equivalente, cf. il teorema 3.3.6.

## 1.2 Macchina di Turing

Introdurremo di seguito uno dei formalismi piú importanti e piú diffusi per esprimere algoritmi: le Macchine di Turing, che ricordano con straordinaria verosimiglianza i comuni elaboratori à la von Neumann, o a programma memorizzabile, cui siamo abituati. Ve ne sono moltissime definizioni, che differiscono per varianti spesso irrilevanti; la definizione originale prevede un esecutore umano e fu introdotta da Alan Turing nel 1936, ben prima che ci fossero dei computer funzionanti (MARK IV, COLOSSUS o ENIAC).

**Definizione 1.2.1** (Macchina di Turing)**.** Si dice Macchina di Turing (in breve MdT) una quadrupla

$$M = (Q, \Sigma, \delta, q_0)$$

dove:

- $Q(\ni q_i)$ è l'insieme *finito* degli *stati*, con $h \notin Q$
  (con lo stato speciale $h$ indicheremo il caso di un arresto "corretto" di un calcolo di $M$)
- $\Sigma(\ni \sigma, \sigma', ...)$ è l'insieme *finito* dei simboli (*alfabeto*) con $\# \in \Sigma$ (*carattere bianco*), $\rhd \in \Sigma$ (*marca di inizio stringa*) e $L, R, - \notin \Sigma$
- $q_0 \in Q$ è lo *stato iniziale*
- $\delta \subseteq (Q \times \Sigma) \times (Q \cup \{h\}) \times \Sigma \times \{L, R, -\}$ è la *relazione di transizione*, tale che

  $$\text{se } (q, \rhd, q', \sigma, D) \in \delta \text{ allora } \sigma = \rhd, D = R$$

  Questa condizione dice che il carattere corrente, ovvero il cursore (vedi la figura piú sotto) non può mai trovarsi a sinistra della marca di inizio stringa, $\rhd$. (Piú intuitivamente se $\delta(q, \rhd) = (q', \sigma, D)$ allora $\sigma = \rhd, D = R$, secondo quanto stipulato qui sotto.)

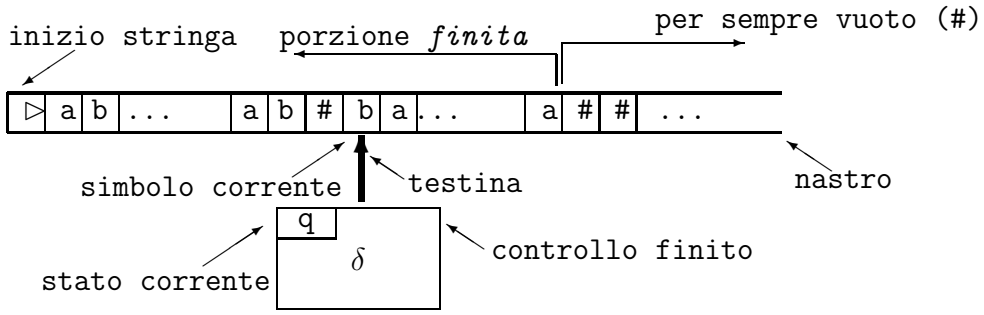
  In questa parte del corso restringeremo la relazione $\delta$ in modo che sia una funzione rispetto ai suoi primi due argomenti, imponendo cioè che, per ogni coppia di quintuple:

  $$(q, \sigma, q', \sigma', D'), (q, \sigma, q'', \sigma'', D'') \in \delta \Rightarrow \ q' = q'', \sigma' = \sigma'', D' = D''$$

  Grazie a questa condizione possiamo scrivere $\delta(q, \sigma) = (q', \sigma', D')$ al posto della quintupla $(q, \sigma, q', \sigma', D')$.

È facile verificare che la condizione (i) dell'idea intuitiva di algoritmo elencata dianzi è soddisfatta, cosí come lo è anche la prima parte della condizione (ii). Esse richiedono che i programmi siano finiti e che le possibili istruzioni, operanti su dati discreti, siano in numero finito. Infatti, ogni macchina ha un numero finito di possibili istruzioni, poiché gli insiemi $Q$ e $\Sigma$ sono finiti; di conseguenza la sua funzione di transizione $\delta$ contiene un numero finito di elementi. Inoltre, i dati su cui opera una MdT sono stringhe $w$ di caratteri appartenenti a $\Sigma$, in simboli $w \in \Sigma^*$. Piú precisamente $\Sigma^*$ contiene la stringa vuota $\epsilon$ e per tutte le stringhe $w \in \Sigma^*$ e tutti i caratteri $a \in \Sigma$ contiene la stringa $a \cdot w$, dove "$\cdot$" è l'operazione associativa (quasi sempre omessa) di giustapposizione tra caratteri (esteso alle stringhe; brevemente, $\Sigma^*$ è il monoide libero generato da $\Sigma$ avente come unica operazione interna associativa "$\cdot$" ($\forall w, w', w" \in \Sigma^*$ vale $w \cdot (w' \cdot w") = (w \cdot w') \cdot w"$) e con identità destra e sinistra $\epsilon$ ($\forall w \in \Sigma^*$ vale $\epsilon \cdot w = w = w \cdot \epsilon$).

Graficamente, una MdT può essere rappresentata dalla figura seguente.




Esiste una variante delle MdT detta *non-deterministica* in cui non si richiede che $\delta$ sia una funzione. Nella terza parte del corso ne daremo una definizione formale, e ricorderemo che ha "la stessa potenza espressiva" della MdT definite sopra. Piú avanti useremo, introducendola in modo intuitivo, anche la variante in cui si hanno più nastri, la cui definizione formale si trova nella definizione 3.2.1. Il suo potere espressivo non cambia, cosí come non cambia quello di tutte le altre varianti, per esempio quelle macchine con un nastro infinito anziché semi-infinito, oppure con piú di un nastro, oppure ancora quelle che possono o scrivere o spostarsi, o hanno altre diavolerie "ragionevoli." Questa robustezza rafforza la sensazione che il modello sia azzeccato.

**Esempio 1.2.2.** [Una macchina che qualcosa fa]

| $q$ | $\sigma$ | $\delta(q,\sigma)$ |
|---|---|---|
| $q_0$ | $\triangleright$ | $q_0, \triangleright, R$ |
| $q_0$ | $\#$ | $h, \#, -$ |
| $q_0$ | $a$ | $q_1, \#, L$ |
| $q_1$ | $\#$ | $q_0, \#, L$ |
| $q_1$ | $a$ | $q_0, a, -$ |

A fianco la tabella che rappresenta la funzione di transizione della Mdt $M = (Q, \Sigma, \delta, q_0)$, con $Q = \{q_0, q_1\}$ e $\Sigma = \{\#, \triangleright, a\}$.

In seguito, nel definire una MdT, ometteremo per brevità la definizione degli insiemi $Q$ e $\Sigma$ degli stati e dei simboli, i cui elementi possono essere dedotti guardando la tabella che definisce la funzione di transizione $\delta$.

Passiamo ora a descrivere la dinamica delle macchine di Turing, cioè il loro comportamento quando operino su una stringa di caratteri memorizzata sul nastro. In altre parole, vogliamo definire cos'è una computazione di una macchina. Intuitivamente, una computazione di una MdT è una successione di configurazioni, ognuna ottenuta dalla precedente in accordo con la definizione della funzione di transizione $\delta$. Ci manca la definizione di configurazione, che verrà data tra poche righe. Per il momento, e in modo del tutto informale, indichiamo con la coppia *Stato/Nastro* una *configurazione istantanea* della macchina; dove *Stato* è lo stato in cui si trova la macchina e *Nastro* è una porzione "abbastanza lunga" e *finita* del nastro che contiene almeno la sua parte non bianca. La seguente tabella rappresenta informalmente una computazione della MdT introdotta nell'esempio 1.2.2, che si arresta con successo in 7 passi. Nelle configurazioni, il carattere sottolineato nella parte *Nastro* sta a indicare la posizione corrente del cursore, o *testina*.

| Configurazione | Azione effettuata |
|---|---|
| $q_0/ \triangleright \#\#a\#a\underline{a}\# \rightarrow$ | "cancella" $a$ cambia stato e vai a sinistra |
| $q_1/ \triangleright \#\#a\#\underline{a}\#\# \rightarrow$ | non scrive, non si sposta, cambia stato |
| $q_0/ \triangleright \#\#a\#\underline{a}\#\# \rightarrow$ | "cancella" $a$ cambia stato e vai a sinistra |
| $q_1/ \triangleright \#\#a\underline{\#}\#\#\# \rightarrow$ | lascia #, cambia stato, va a sinistra |
| $q_0/ \triangleright \#\#\underline{a}\#\#\#\# \rightarrow$ | "cancella" $a$ cambia stato e vai a sinistra |
| $q_1/ \triangleright \#\underline{\#}\#\#\#\# \rightarrow$ | lascia #, cambia stato, va a sinistra |
| $q_0/ \triangleright \underline{\#}\#\#\#\#\#\# \rightarrow$ | si arresta |
| $h/ \triangleright \underline{\#}\#\#\#\#\#\#$ | |

Piú precisamente, una *configurazione* $C$ di una MdT è una quadrupla

$$(q, u, \sigma, v) \in (Q \cup \{h\}) \times \Sigma^* \times \Sigma \times \Sigma^F$$

dove $q$ è lo stato corrente, $u$ è la stringa di caratteri a sinistra del simbolo corrente $\sigma$ e $v$ è quella dei caratteri alla sua destra. Per semplicità scriveremo

$(q, u\underline{\sigma}v)$ al posto di $(q, u, \sigma, v)$, cosí come abbiam già fatto sopra. Poiché la stringa $u\sigma v$ deve essere finita (e poiché per ora) [2] non ci interessa considerare i simboli $\#$ a destra del simbolo $\sigma_n \neq \#$ piú a destra nel nastro (ovvero della porzione $v$), indichiamo con $\epsilon$ la stringa vuota e conveniamo che $\# \cdot \epsilon = \epsilon$ e $\sigma \cdot \epsilon = \sigma$, se $\sigma \neq \#$. Definiamo allora $\Sigma^F = \Sigma^* \cdot (\Sigma \setminus \{\#\}) \cup \{\epsilon\}$. Quindi scriviamo $v = \sigma_0\sigma_1 \ldots \sigma_n$, con $\sigma_n \neq \#$, al posto della stringa infinita $\sigma_0\sigma_1 \ldots \sigma_n\#\# \ldots \# \ldots$. Si noti tuttavia che può benissimo darsi che $\sigma_i = \#$ per qualche $i < n$ e anche che il carattere corrente $\sigma$ può essere $\#$; inoltre la stringa $u$ può essere vuota solo quando il carattere corrente $\sigma$ è la marca di inizio stringa $\triangleright$, per la seconda condizione sulla funzione di transizione $\delta$. Con questa convenzione ogni stringa è *finita*.

Il frammento iniziale della computazione di prima viene rappresentato come:

$$(q_0, \triangleright\#\#a\#a, a, \epsilon) \to (q_1, \triangleright\#\#a\#, a, \epsilon) \to$$
$$(q_0, \triangleright, \#\#a\#, a, \epsilon) \to (q_1, \triangleright\#\#a, \#, \epsilon) \to \ldots$$

Un altro esempio di computazione della macchina definita nell'es. 1.2.2 è:

$$(q_0, \triangleright\#, a, a) \to (q_1, \triangleright, \#, \#a) \to (q_0, \epsilon, \triangleright, \#\#a) \to$$
$$(q_0, \triangleright, \#, \#a) \to (h, \triangleright, \#, \#a)$$

Per non eccedere nella pignoleria, non sempre utilizzeremo la versione di configurazione nella forma definita sopra, ma ci riterremo liberi di scrivere $(q, w)$ quando non interessi sapere dove si trova il cursore (v. la definizione di computazione piú sotto), o di usare altre convenzioni quando il loro significato sia chiaro dal contesto.

Formalmente un *passo di computazione* è definito per casi nel modo seguente, intendendo che le quadruple che vi appaiono siano configurazioni (ricordandosi che $q \neq h$ e usando $q'$ per indicare, oltre a uno stato in $Q$, anche $h$; indicando con $a, b, c$ elementi generici di $\Sigma$; e ricordando inoltre, soprattutto nel caso (ii), che se $b = \#$ e $v = \epsilon$ allora $bv = \epsilon$ e infine che nel caso (iii) se $a = \triangleright$ allora anche $b = \triangleright$):

| | | | |
|---|---|---|---|
| i) | | $(q, u\underline{a}v) \to (q', u\underline{b}v)$ | se $\delta(q, a) = (q', b, -)$ |
| ii) | | $(q, uc\underline{a}v) \to (q', u\underline{c}bv)$ | se $\delta(q, a) = (q', b, L)$ |
| iii) | (a) | $(q, u\underline{a}cv) \to (q', ub\underline{c}v)$ | se $\delta(q, a) = (q', b, R)$ |
| | (b) | $(q, u\underline{a}) \to (q', ub\underline{\#})$ | se $\delta(q, a) = (q', b, R)$ |

Ciascun passo ha un effetto limitato sulle configurazioni, come richiesto dalla seconda parte del punto (ii) nella idea intuitiva di algoritmo. Inoltre, un

[2] Nella terza parte del corso terremo traccia di tutte le caselle visitate dalla MdT, ottenendo cosí una stima dello spazio necessario per quella computazione.

singolo passo dipende *unicamente* da *un solo simbolo*, quello corrente,[3] e da *un solo stato*, quello corrente (cf. i punti (iii) e (iv) richiesti dall'intuizione). Si noti che sia per determinare la regola da applicare che per applicarla diamo per primitiva la capacità dell'esecutore di ricorrere al "pattern matching." Una *computazione* è una successione finita di passi

$$(q_0, w) \to^* (q', w')$$

dove $\to^*$ è la chiusura riflessiva e transitiva di $\to$. Come d'abitudine, se vi sono $n$ passi, la computazione è lunga $n$ e la scriveremo così:

$$(q_0, w) \to^n (q', w').$$

Diremo invece che la computazione $(q_0, w) \to^* (q', w')$ *termina* (converge, $\downarrow$) su $w$ sse $q' = h$, e che *non termina* (diverge, $\uparrow$) sse per ogni $q', w'$ tali che $(q_0, w) \to^* (q', w')$ esistono $q'', w''$ tali che $(q', w') \to (q'', w'')$.

Se fosse necessario precisare che la computazione in esame è una di quelle di una particolare macchina $M$, scriveremo $\to^*_M$; con $M(w)$ esprimeremo che la macchina $M$ inizia la sua computazione dalla configurazione $(q_0, \underline{\triangleright} w)$, cioè avendo come dato iniziale la stringa $w$, ovvero che *applichiamo* $M$ a $w$.

*Domanda* 1.2.3. C'è un limite ai passi di computazione o allo spazio necessario a contenerne i dati (punto 5 della definizione intuitiva di algoritmo)? **NO**, come si vede dal seguente esempio.

**Esempio 1.2.4.** [Una macchina che non converge per nessun ingresso]

| $q$ | $\sigma$ | $\delta$ |
|---|---|---|
| $q_0$ | $\triangleright$ | $q_0, \triangleright, R$ |
| $q_0$ | $a$ | $q_0, a, R$ |
| $q_0$ | $\#$ | $q_0, \#, R$ |

Un esempio di computazione non terminante della macchina a fianco è $(q_0, \underline{\triangleright} a\#a\#) \to (q_0, \triangleright \underline{a}\#a\#) \to^* (q_0, \triangleright a\#a\# \ldots \underline{\#}\#) \to \ldots$

Ovviamente, le macchine di Turing sono state introdotte per formalizzare la nozione di calcolo. Lo faremo tra poco e tuttavia negli esempi che seguono ci sentiamo liberi di usare le parole "calcolare" e "decidere".

**Esempio 1.2.5.** [Quanto fa $n + m$? con una benevola interpretazione ...] Definiamo una MdT che intuitivamente calcola $n + m$ dove $n$ ed $m$ sono interi positivi rappresentati in notazione unaria con il simbolo | e separati dal simbolo + (stipuliamo che il nastro iniziale abbia sempre questa forma). In notazione unaria, vi sono tanti | giustapposti quanto è il numero da rappresentare; cioè $n$ è rappresentato da $|^n$; ad esempio $|^3 = |||$ sta per 3.

[3] Si noti che l'unica maniera per ispezionare una parte non finita del nastro sarebbe quella di accedere a *tutto* il nastro a destra del cursore.

| $q$ | $\sigma$ | $\delta$ |
|---|---|---|
| $q_0$ | $\rhd$ | $q_0, \rhd, R$ |
| $q_0$ | $\|$ | $q_0, \|, R$ |
| $q_0$ | $+$ | $q_1, \|, R$ |
| $q_1$ | $\|$ | $q_1, \|, R$ |
| $q_1$ | $\#$ | $q_2, \#, L$ |
| $q_2$ | $\|$ | $h, \#, -$ |

La computazione per il calcolo di $1 + 2$ è la seguente:

$(q_0, \trianglerighteq \mid + \mid\mid) \to (q_0, \rhd\underline{\mid} + \mid\mid) \to (q_0, \rhd \mid \underline{+} \mid\mid)$
$\to (q_1, \rhd \mid\mid \underline{\mid} \mid) \to (q_1, \rhd \mid\mid\mid \underline{\mid}) \to (q_1, \rhd \mid\mid\mid\mid \underline{\#})$
$\to (q_2, \rhd \mid\mid\mid \underline{\mid}) \to (h, \rhd \mid\mid\mid \underline{\#})$

**Esempio 1.2.6.** [cosa accomuna "il burino con i rubli" ad "accavallavacca"?] La macchina in figura 1.1 si arresta con il simbolo *SI* nella casella scritta più a destra e il cursore sul primo carattere del nastro bianco se ha in ingresso una stringa palindroma su $\{a, b\}^*$; se non lo è, la macchina va in stallo entrando in uno stato speciale $q_N$. Non sarebbe difficile estendere questa macchina in modo che si arrestasse nello stato $h$ anche in questo caso, lasciando però sul nastro il carattere *NO*. Avremmo cosí rappresentato un algoritmo che *decide* se la stringa in ingresso appartiene all'insieme delle stringhe palindrome o meno. Il lettore avrà certamente notato una leggera differenza di stile tra questa e la macchina precedente che invece *calcola* una funzione.

Nella tabella che descrive la funzione di transizione $\delta$, abbreviamo con $(q_{a/b}, \sigma, q'_{a/b}, \sigma', D)$ le due quintuple $(q_a, \sigma, q'_a, \sigma', D)$ e $(q_b, \sigma, q'_b, \sigma', D)$ (cioè $q_{a/b}$ non è uno stato singolo, ma rappresenta gli stati $q_a$ e $q_b$; analogamente per $q'_{a/b}$ e per $q'_a$, $q'_b$). Al suo fianco una computazione che si arresta con successo, perché *aba* è palindroma. Si noti che la marca di inizio stringa $\rhd$ si sposta a destra, ma non viene mai oltrepassata. Non è difficile modificare la funzione di transizione in modo che ciò non accada: basta cancellare il simbolo sostituito da $\rhd$ e spostare l'intera stringa residua a sinistra di una casella.

Vediamo adesso un altro esempio di computazione che si arresta con successo, perché il suo argomento *abba* è una striga palindroma. Per comprendere meglio il comportamento della macchina, abbiamo raggruppato i suoi passi in fasi, che operano in maniera "omologa". Inoltre, in questo esempio e in altri nel capitolo, abbiamo mantenuto traccia nelle configurazioni di tutte le posizioni che la macchina visita, per valutare, nella terza parte del corso, lo spazio necessario alle computazioni di questa macchina.

$(q_0, \trianglerighteq abba\#) \to$
$(q_o, \rhd\underline{a}bba\#) \to (q_a, \rhd \rhd \underline{b}ba\#) \to (q_a, \rhd \rhd b\underline{b}a\#) \to (q_a, \rhd \rhd bb\underline{a}\#) \to (q_a, \rhd \rhd bba\underline{\#}) \to$
$(q'_a, \rhd \rhd bb\underline{a}\#) \to (q_1, \rhd \rhd b\underline{b}\#\#) \to (q_1, \rhd \rhd \underline{b}b\#\#) \to (q_1, \rhd \trianglerighteq bb\#\#) \to$
$(q_0, \rhd \rhd \underline{b}b\#\#) \to (q_b, \rhd \rhd \rhd\underline{b}\#\#) \to (q_b, \rhd \rhd \rhd b\underline{\#}\#\#) \to$
$(q'_b, \rhd \rhd \rhd\underline{b}\#\#) \to (q_1, \rhd \rhd \trianglerighteq\#\#\#) \to (q_0, \rhd \rhd \rhd\underline{\#}\#\#) \to$
$(q_2, \rhd \rhd \rhd\underline{\#}\#\#) \to (h, \rhd \rhd \rhd SI\underline{\#}\#)$

| $q$ | $\sigma$ | $\delta$ |
|---|---|---|
| $q_0$ | $\rhd$ | $q_0, \rhd, R$ |
| $q_0$ | $a$ | $q_a, \rhd, R$ |
| $q_0$ | $b$ | $q_b, \rhd, R$ |
| $q_0$ | $\#$ | $q_2, \#, -$ |
| $q_{a/b}$ | $a$ | $q_{a/b}, a, R$ |
| $q_{a/b}$ | $b$ | $q_{a/b}, b, R$ |
| $q_{a/b}$ | $\#$ | $q'_{a/b}, \#, L$ |
| $q'_a$ | $\rhd$ | $q_2, \rhd, R$ |
| $q'_a$ | $a$ | $q_1, \#, L$ |
| $q'_a$ | $b$ | $q_N, b, -$ |
| $q'_b$ | $\rhd$ | $q_2, \rhd, R$ |
| $q'_b$ | $a$ | $q_N, a, -$ |
| $q'_b$ | $b$ | $q_1, \#, L$ |
| $q_1$ | $\rhd$ | $q_0, \rhd, R$ |
| $q_1$ | $a$ | $q_1, a, L$ |
| $q_1$ | $b$ | $q_1, b, L$ |
| $q_2$ | $\#$ | $h, SI, R$ |

La sua computazione su $aba$ è la seguente:

$(q_0, \trianglerighteq aba\#) \to (q_o, \rhd \underline{a} ba\#) \to$
$(q_a, \rhd \rhd \underline{b} a\#) \to (q_a, \rhd \rhd b\underline{a}\#) \to$
$(q_a, \rhd \rhd ba\underline{\#}) \to (q'_a, \rhd \rhd b\underline{a}\#) \to$
$(q_1, \rhd \rhd \underline{b}\#\#) \to (q_1, \rhd \trianglerighteq b\#\#) \to$
$(q_0, \rhd \rhd \underline{b}\#\#) \to (q_b, \rhd \rhd \rhd \underline{\#}\#) \to$
$(q'_b, \rhd \rhd \trianglerighteq \#\#) \to (q_2, \rhd \rhd \rhd \underline{\#}\#) \to$
$(h, \rhd \rhd \rhd SI\underline{\#})$

Figura 1.1: Una macchina che "decide" se una stringa è palindroma

La seguente computazione si arresta perché la funzione $\delta$ non è definita per lo stato $q_N$; quindi, non lascia la macchina in una configurazione terminale e riflette il fatto che la stringa $ba$ non è palindroma.

$(q_0, \trianglerighteq ba\#) \to (q_0, \rhd \underline{b} a\#) \to (q_b, \rhd \rhd \underline{a}\#) \to (q_b, \rhd \rhd a\underline{\#}) \to (q'_b, \rhd \rhd \underline{a}\#) \to$
$(q_N, \rhd \rhd \underline{a}\#)$

Come detto sopra, non è difficile definire delle quintuple che, a partire dallo stato $q_N$, cancellino la parte di nastro contenente simboli diversi dal $\#$ e dal $\rhd$, scrivano nella prima casella utile $NO$ in modo che la macchina si arresti con successo.

Nel seguito, una situazione di arresto "anomalo", come quella appena vista verrà talvolta chiamata *stallo*.

Complichiamoci adesso un po' la vita e sostituiamo alla quintupla $\delta(q'_a, b) = (q_N, b, -)$ la quintupla $\delta(q'_a, b) = (q'_a, b, -)$ e similmente alla quintupla $\delta(q'_b, a) = (q_N, a, -)$ sostituiamo $\delta(q'_b, a) = (q'_b, a, -)$. È immediato verificare che la macchina, applicata a una stringa non palindroma, non si arresterà mai.

## 1.3 Linguaggi *FOR* e *WHILE*

Introduciamo adesso un altro formalismo per esprimere funzioni, certamente piú familiare: un linguaggio di programmazione, o meglio il suo nucleo, in forma essenziale. Si tratta di un linguaggio imperativo semplicissimo, che opera solo sui numeri naturali e sui valori di verità, con strutture di controllo che sono presenti in ogni linguaggio di programmazione.

### Sintassi (astratta)

$$
\begin{array}{lcll}
E & \to & n \mid x \mid E_1 + E_2 \mid E_1 \times E_2 \mid E_1 - E_2 & \text{Espr. Aritmetiche} \\
B & \to & b \mid E_1 < E_2 \mid \neg B \mid B_1 \vee B_2 & \text{Espr. Booleane} \\
C & \to & \texttt{skip} \mid x := E \mid C_1; C_2 \mid \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2 \mid & \text{Comandi} \\
 & & \texttt{for } x = E_1 \texttt{ to } E_2 \texttt{ do } C \mid \texttt{while } B \texttt{ do } C &
\end{array}
$$

dove $n \in \mathbb{N}$ (i numeri naturali), $x \in \mathrm{Var}$ (un insieme numerabile di variabili), $b \in \mathrm{Bool} = \{tt, ff\}$ (i valori di verità).[4]

Chiameremo *WHILE* il linguaggio definito dalla grammatica BNF definita sopra [5]; chiameremo invece *FOR* il linguaggio risultante dall'ommissione del comando `while` $B$ `do` $C$ nella definizione della sintassi.

### Semantica (ovvero calcolare è dimostrare)

Prima di definire la dinamica del nostro linguaggio abbiamo bisogno di un paio di nozioni ausiliarie. Rappresentiamo la memoria mediante una funzione (da sottoinsiemi *finiti* di Var nei numeri naturali, tanto in un programma ci possono essere solo un numero finito di variabili)

$$\sigma : \mathrm{Var} \to \mathbb{N}$$

e definiamo l'operazione di *aggiornamento* tramite la funzione, o meglio il funzionale a tre argomenti

$$-[-/-] : (\mathrm{Var} \to \mathbb{N}) \times \mathbb{N} \times \mathrm{Var} \to (\mathrm{Var} \to \mathbb{N})$$

[4] Poiché non vi sono comandi di lettura né di scrittura su periferiche, assumeremo nel seguito che i programmi abbiano già la memoria inizializzata con i dati di ingresso (cioè che abbiamo una $\sigma$ iniziale, vedi sotto) e che i risultati finali siano leggibili in memoria.

[5] Di solito si indica con *WHILE* il linguaggio definito dalla grammatica di cui sopra, privata della produzione relativa al comando `for` $x = E_1$ `to` $E_2$ `do` $C$. Avere o non avere comandi di questo tipo non crea alcun problema, in quanto essi sono "esprimibili" mediante un oppotuno comando di tipo `while` B `do` C.

definito come $\sigma[n/x](y) = \begin{cases} n & \text{se } y = x \\ \sigma(y) & \text{altrimenti} \end{cases}$

Il significato, o *semantica* di un'espressione aritmetica è data dalla seguente funzione, definita ovunque. Seguendo la tradizione scriveremo il suo argomento principale, l'espressione aritmetica, tra le parentesi $[\![$ e $]\!]$, cui viene giustapposto il secondo argomento, cioè la memoria in cui l'espressione va valutata.

$$\mathcal{E}[\![-]\!]- : E \times (\mathrm{Var} \to \mathbb{N}) \to \mathbb{N}$$

$$\begin{array}{lcl} \mathcal{E}[\![n]\!]\sigma & = & n \\ \mathcal{E}[\![x]\!]\sigma & = & \sigma(x) \\ \mathcal{E}[\![E_1 + E_2]\!]\sigma & = & \mathcal{E}[\![E_1]\!]\sigma \textit{ piú } \mathcal{E}[\![E_2]\!]\sigma \\ \mathcal{E}[\![E_1 \times E_2]\!]\sigma & = & \mathcal{E}[\![E_1]\!]\sigma \textit{ per } \mathcal{E}[\![E_2]\!]\sigma \\ \mathcal{E}[\![E_1 - E_2]\!]\sigma & = & \mathcal{E}[\![E_1]\!]\sigma \textit{ meno } \mathcal{E}[\![E_2]\!]\sigma \end{array}$$

dove *piú*, *per*, *meno*, sono "vere" funzioni da coppie di numeri naturali a numeri naturali, a differenza di $+$, $\times$ $-$ che sono solo simboli (*tokens*) del nostro linguaggio di programmazione. Poiché i nostri dati sono solo numeri naturali, la funzione *meno* che impieghiamo è la funzione, di solito chiamata *meno limitato*, che si comporta come aspettato quando il minuendo è maggiore o uguale al sottraendo e restituisce 0 altrimenti.

Invece, nella prima equazione, non abbiamo fatto distinzione tra i *numerali* e i *numeri naturali* e li abbiamo rappresentati con gli stessi simboli; pur non essendo del tutto corretto, ciò viene comunemente fatto e non dovrebbe creare problemi perché di solito è chiaro quando $n$ rappresenta un numero e quando un simbolo del linguaggio.

**Esempio 1.3.1.** Si debba valutare l'espressione [6]

$$x \times 2 - ((y - 7) + 1)$$

nella memoria $\sigma$ tale che

$$\sigma(x) = 3, \sigma(y) = 5.$$

[6]Le parentesi sono state scritte solo per chiarezza, infatti bisogna immaginare che l'espressione sia rappresentata in forma astratta come un albero e che quindi non vi sia ambiguità nella sua interpretazione; per la stessa ragione, $x \times 2$ non è stata racchiusa tra parentesi.

Abbiamo allora

$$
\begin{array}{l}
\mathcal{E}[\![x \times 2 - ((y-7)+1)]\!]\sigma = \\
\mathcal{E}[\![x \times 2]\!]\sigma \; \mathit{meno} \; \mathcal{E}[\![(y-7)+1]\!]\sigma = \\
(\mathcal{E}[\![x]\!]\sigma \; \mathit{per} \; 2) \; \mathit{meno} \; \mathcal{E}[\![(y-7)+1]\!]\sigma = \\
(\sigma(x) \; \mathit{per} \; 2) \; \mathit{meno} \; \mathcal{E}[\![(y-7)+1]\!]\sigma = \\
(3 \; \mathit{per} \; 2) \; \mathit{meno} \; \mathcal{E}[\![(y-7)+1]\!]\sigma = \\
6 \; \mathit{meno} \; (\mathcal{E}[\![y-7]\!]\sigma \; \mathit{piú} \; 1) = \\
6 \; \mathit{meno} \; ((\mathcal{E}[\![y]\!]\sigma \; \mathit{meno} \; 7) \; \mathit{piú} \; 1) = \\
6 \; \mathit{meno} \; ((\sigma(y) \; \mathit{meno} \; 7) \; \mathit{piú} \; 1) = \\
6 \; \mathit{meno} \; ((5 \; \mathit{meno} \; 7) \; \mathit{piú} \; 1) = \\
6 \; \mathit{meno} \; (0 \; \mathit{piú} \; 1) = \\
6 \; \mathit{meno} \; 1 = 5
\end{array}
$$

La semantica di un'espressione booleana è data dalla seguente funzione, ovunque definita, che usa la $\mathcal{E}$ appena introdotta.

$$\mathcal{B}[\![-]\!]- : B \times (\mathrm{Var} \to \mathbb{N}) \to \mathit{Bool}$$

$$
\begin{array}{lcl}
\mathcal{B}[\![t]\!]\sigma & = & \mathit{tt} \\
\mathcal{B}[\![f]\!]\sigma & = & \mathit{ff} \\
\mathcal{B}[\![E_1 < E_2]\!]\sigma & = & \mathcal{E}[\![E_1]\!]\sigma \; \mathit{minore} \; \mathcal{E}[\![E_2]\!]\sigma \\
\mathcal{B}[\![\neg B]\!]\sigma & = & \mathit{not} \; \mathcal{B}[\![B]\!]\sigma \\
\mathcal{B}[\![B_1 \vee B_2]\!]\sigma & = & \mathcal{B}[\![B_1]\!]\sigma \; \mathit{or} \; \mathcal{B}[\![B_2]\!]\sigma
\end{array}
$$

dove *minore*, *not* e *or* sono "vere" funzioni.

Lo stile di definizione seguito per dar semantica alle espressioni va sotto il nome di stile *denotazionale* e si propone di associare una funzione a ciascun operatore, o piú in generale, a ciascun costrutto del linguaggio. La definizione di un costrutto composto viene data in termini dei suoi componenti, e per questa proprietà a volte questo stile viene anche detto *composizionale*. Adesso daremo la semantica dei comandi usando un approccio *operazionale*, guidato dalla sintassi. In effetti, andremo a definire una sorta di macchina astratta, che procede per passi discreti, apportando modifiche discrete sulle sue configurazioni, o stati, che possiamo immaginare come coppie $\langle \mathit{comando}, \sigma \rangle$. Questa macchina astratta, che definisce in modo intensionale la semantica di un linguaggio di programmazione o di un sistema, va sotto il nome di *sistema di transizioni* e formalmente è una coppia

$$(\Gamma, \to)$$

dove

- $\Gamma$ è la classe delle *configurazioni* (nel nostro caso la classe delle coppie $\langle c \in C, \sigma \rangle$ dove $\sigma$ è definita per ogni variabile che appaia in *comando*; la coppia con il comando $c$ vuoto ($\neq$ `skip`) è abbreviata da $\sigma$);

- $\rightarrow \subseteq \Gamma \times \Gamma$ è una funzione, detta di *transizione*.

Noi seguiremo un approccio detto SOS (Structural Operational Semantics) (o *small-step*), in cui ciascuna transizione

$$\langle c, \sigma \rangle \rightarrow \langle c', \sigma' \rangle$$

rappresenta un singolo passo di computazione, ovvero il fatto che la macchina ha attraversato lo stato $\langle c, \sigma \rangle$ e transisce nello stato $\langle c', \sigma' \rangle$ a causa di una sua certa attività. [7]

In maniera del tutto simile a quanto fatto con le macchine di Turing, si può allora definire una *computazione* come la chiusura riflessiva e transitiva della funzione di transizione, formalmente, una *computazione* è

$$\langle c, \sigma \rangle \rightarrow^* \langle c', \sigma' \rangle$$

Una computazione termina con successo, o converge, se $\langle c, \sigma \rangle \rightarrow^* \sigma'$.

Si noti che la semantica operazionale di un programma è allora data dal grafo $(\Gamma, \rightarrow)$, mentre le computazioni danno una descrizione dei passi necessari al calcolo con granularità fine quanto si voglia.

In questo approccio, l'attività svolta dalla macchina quando transisce da una configurazione nella successiva viene a volte resa esplicita, etichettando la funzione di transizione e ottenendo cosí i *sistemi di transizioni etichettati*.[8] Un'ulteriore estensione piuttosto comune riguarda l'arricchimento della coppia $(\Gamma, \rightarrow)$ con un insieme di configurazioni *terminali*, ovvero di stati finali (a volte si aggiunge alla coppia anche una configurazione *iniziale*, da cui si assume partano tutte le computazioni). Nel nostro caso, il sistema di transizioni che useremo avrà come configurazioni finali le coppie formate da un comando vuoto e una memoria $\sigma$, che abbiamo già abbreviato con $\sigma$.

È evidente come gli stili di definizione denotazionale e operazionale siano differenti, in quanto non c'è modo immediato di comporre il "risultato" di un'esecuzione con quello di un'altra esecuzione. Tuttavia, anche l'approccio

[7] Un approccio alternativo consiste nel definire una semantica *naturale* o *big-step*, in cui le transizioni hanno tutte la forma

$$\langle c \in C, \sigma \rangle \Rightarrow \sigma'$$

cioè $\sigma'$ memorizza le modifiche prodotte dall'*intera* esecuzione del comando $c$ nella memoria iniziale $\sigma$. Un'intera computazione viene allora vista come la deduzione della transizione in questione. Potremmo però trovarci nell'impossibilità di dimostare l'esistenza di una transizione per uno stato: in questo caso avremmo o una situazione di stallo, o una situazione di non-terminazione della computazione, quando la deduzione non sia finita.

[8] Ovviamente questo può venir fatto anche nell'approccio naturale, componendo le etichette e rappresentando cosí le computazioni attraverso una sequenza di etichette.

operazionale è in larga misura composizionale, per il modo che seguiamo nel definire la classe delle transizioni. Infatti, di seguito introduciamo un insieme di assiomi e regole di inferenza, che inducendo sulla sintassi dei comandi, ci permettono di dedurre tutte e sole le transizioni (ovvero prendiamo la *minima* classe che contiene tutte le instanze degli assiomi ed è chiusa rispetto alle regole di inferenza).

$$\frac{-}{\langle \texttt{skip}, \sigma\rangle \to \sigma} \qquad \frac{-}{\langle x := E, \sigma\rangle \to \sigma[n/x]}, \text{ se } \mathcal{E}[\![E]\!]\sigma = n$$

$$\frac{\langle C_1, \sigma\rangle \to \langle C_1', \sigma'\rangle}{\langle C_1; C_2, \sigma\rangle \to \langle C_1'; C_2, \sigma'\rangle} \qquad \frac{\langle C_1, \sigma\rangle \to \sigma'}{\langle C_1; C_2, \sigma\rangle \to \langle C_2, \sigma'\rangle}$$

$$\frac{-}{\langle \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2, \sigma\rangle \to \langle C_1, \sigma\rangle}, \text{ se } \mathcal{B}[\![B]\!]\sigma = tt$$

$$\frac{-}{\langle \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2, \sigma\rangle \to \langle C_2, \sigma\rangle}, \text{ se } \mathcal{B}[\![B]\!]\sigma = ff$$

$$\frac{-}{\langle \texttt{for } i = E_1 \texttt{ to } E_2 \texttt{ do } C, \sigma\rangle \to \langle i := n_1;\ C;\ \texttt{for } i = n_1 + 1 \texttt{ to } n_2 \texttt{ do } C, \sigma\rangle}$$
$$\text{se } \mathcal{B}[\![E_2 < E_1]\!]\sigma = ff \wedge \mathcal{E}[\![E_1]\!]\sigma = n_1 \wedge \mathcal{E}[\![E_2]\!]\sigma = n_2$$

$$\frac{-}{\langle \texttt{for } i = E_1 \texttt{ to } E_2 \texttt{ do } C, \sigma\rangle \to \sigma} \qquad \text{se } \mathcal{B}[\![E_2 < E_1]\!]\sigma = tt$$

$$\frac{-}{\langle \texttt{while } B \texttt{ do } C, \sigma\rangle \to \langle \texttt{if } B \texttt{ then } C;\ \texttt{while } B \texttt{ do } C \texttt{ else skip}, \sigma\rangle}$$

Commenti alle definizioni sopra. P.e. sul $\texttt{for } i = n_1 + 1 \texttt{ to } n_2 \texttt{ do } C$ che mescola sintassi $E$ con semantica $n_1 + 1$; da aggiustare con gli assegnamenti $z := E_1; w := E_2$, con $z, w \notin fn(E1, E_2, C)$.

**Esempio 1.3.2.** Esempio di deduzione PAG 12 appunti miei

*Domanda* 1.3.3. Ci sono computazioni: che non terminano in *FOR*? e in *WHILE*?

Suggerimento 1. Quante volte viene eseguito il corpo $C$ nella configurazione $\langle \texttt{for } i = E_1 \texttt{ to } E_2 \texttt{ do } C, \sigma\rangle$? È importante che il comando $C$ non modifichi l'indice $i$ di iterazione,[9] come viene spesso richiesto nei linguaggi di programmazione?
Suggerimento 2. Si consideri la seguente configurazione e si determini la sua evoluzione: $\langle \texttt{while } tt \texttt{ do skip}, \sigma\rangle$.

[9]Si noti che le regole di inferenza date non garantiscono questo fatto, ma solo che il numero di iterazioni non viene modificato anche se ci fossero assegnamenti alla variabile di controllo $i$ all'interno di $C$.

## 1.4 Problemi e Funzioni

Ma cos'è un problema? Qualcosa del tipo: una domanda, con alcuni parametri da assegnare, ovvero una classe (non limitata) di domande, ciascuna delle quali vorremmo che avesse una risposta esatta e finita. Nel nostro caso l'essenza di un problema è formalizzata come:

- calcolare una funzione, oppure
- decidere l'appartenenza a un dato insieme.

Un esempio banale di problema è la domanda: "qual è il massimo comun divisore di $x$ e $y$?" Meno interessante è il problema quando le sue variabili $x$ e $y$ siano state rimpiazzate da un valore, per esempio "qual è il massimo comun divisore di 42 e 56?". Spesso questa semplificazione verrà chiamata *caso* del problema, oppure *istanza*, con orribile anglismo.

Per esempio, la MdT per la somma [10] introdotta nell'esempio 1.2.5 calcola la funzione $+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ (quando applicata alla stringa iniziale $|^n + |^m$); la MdT per le stringhe palindrome dell'esempio 1.2.6 se termina con successo dice che la stringa corrispondente al dato iniziale appartiene a $\{w \in \{a, b\}^* \mid w = w^R\}$ (con $w^R$ si intende la stringa $w$ letta da destra a sinistra, quindi $w$ può essere letta indifferentemente da destra a sinistra o viceversa); se va in stallo dice che non vi appartiene. Si noti tuttavia che nell'ultima variante introdotta ciò non è piú vero: la macchina non si arresta mai se la stringa non è palindroma e quindi abbiamo un algoritmo che dà risultato *solo se* il dato iniziale è una stringa palindroma. Nel primo caso, abbiamo una funzione che opera sui naturali, o meglio su una loro codifica nell'alfabeto $\{|, +, \#\}$; nel secondo caso si considerano insiemi di stringhe.

Da qui al capitolo 1.10 considereremo gli algoritmi come modi per calcolare funzioni, o per rappresentarle in modo finito, per cui dobbiamo metterci d'accordo sul termine *funzione*.

**Definizione 1.4.1** (Funzione totale)**.** $f : A \to B$, sottoinsieme di $A \times B$ è una *funzione totale* se e solamente se

i) $\forall a \in A \ \exists b \in B : (a, b) \in f$, (*la funzione è definita ovunque*)

ii) $(a, b), (a, c) \in f \Rightarrow b = c$ (*unicità*)

**Esempio 1.4.2.** *doppio* : $\mathbb{N} \to \mathbb{N}$ è definita (di solito) come l'insieme delle coppie $doppio = \{(0, 0), (1, 2), (2, 4), \ldots\}$; da cui sappiamo per esempio che $doppio(5) = 10$. Qualche volta, per far prima (!?) scriviamo $doppio(n) =$

[10] Qui e dopo usiamo le interpretazioni usate alla fine del capitolo 1.2.

$n + n$ oppure ancora $doppio(n) = 2 \times n$, dando per note le funzioni + o $\times$ e anche qualche altra cosa (cf. piú avanti la definizione delle funzioni ricorsive).

Tuttavia, non tutte le funzioni sono totali. Ecco un esempio.

**Esempio 1.4.3.** Definiamo, come si fa usualmente, la funzione $mezzo : \mathbb{N} \to \mathbb{N}$, come l'insieme delle coppie $mezzo = \{(0,0), (2,1), (4,2), \cdots\}$ oppure in maniera piú concisa come $mezzo(n) = n/2$. Questa funzione è ovviamente parziale, in quanto è definita solo se $n$ è pari, mentre non ha valore (in $\mathbb{N}$) se $n$ è dispari.

Certamente la funzione *mezzo* è una funzione interessante, e di funzioni parziali ancor piú interessanti di questa se ne incontrano a bizzeffe. Inoltre, vedremo piú avanti che siamo *obbligati* ad avere funzioni che non sono ovunque definite, perciò introduciamo le funzioni parziali.

**Definizione 1.4.4** (Funzione parziale)**.**
$f : A \to B$ è una funzione *parziale* se è un sottoinsieme di $A \times B$ tale che

b) $(a,b), (a,c) \in f \Rightarrow b = c$ (*unicità*), ovvero

b′) esiste al più un $b \in B$ tale che $f(a) = b$,

e quindi non si richiede che $f$ sia ovunque definita.

**Notazione** Data una funzione $f : A \to B$

- diremo che $f$ è *definita* o *converge* su $a$ (in simboli $f(a)\downarrow$) se $\exists b$ tale che $(a,b) \in f$ (cioè $f(a) = b$).
- altrimenti se $\nexists b$ tale che $(a,b) \in f$, diremo che $f$ non è definita o *diverge* su $a$ ($f(a)\uparrow$).

Chiamiamo anche

- *dominio* di $f$ l'insieme $\{a \in A \mid f(a)\downarrow\}$ ($= A$ se e solo se $f$ è totale).
- *codominio* di $f$ l'insieme $B$.
- *immagine* (o *range*) di $f$ l'insieme $\{b \in B \mid \exists a \in A : f(a) = b\}$.

Infine ricordiamo che

- $f$ è *iniettiva* se e solamente se $\forall a, b \in A.\ a \neq b$ implica $f(a) \neq f(b)$ (a volte nella terminologia nord-americana si dice che $f$ è uno-a-uno)
- $f$ è *surgettiva* se e solamente se $\forall b \in B.\ \exists a\ \ \in A$ tale che $f(a) = b$ (ovvero se l'immagine e il codominio di $f$ coincidono)
- $f$ è *biunivoca* se e solamente se è iniettiva e surgettiva.

(Si noti che una funzione iniettiva è invertibile sull'immagine.)

Siamo pronti a dire quando una funzione è calcolabile da una MdT e quando da un programma *WHILE*.

**Definizione 1.4.5.** Siano $\Sigma, \Sigma_0, \Sigma_1$ alfabeti, con $\#, \triangleright \notin \Sigma_0 \cup \Sigma_1$, $\Sigma_0 \cup \Sigma_1 \subset \Sigma$, e $f : \Sigma_0^* \to \Sigma_1^*$ una funzione. Allora si dice che una MdT $M = (Q, \Sigma, \delta, q_0)$ *calcola* $f$, oppure che $f$ è *Turing-calcolabile* o semplicemente *T-calcolabile*, se e solamente se

$$\forall w \in \Sigma_0^* : f(w) = z \text{ se e solamente se } (q_0, \trianglerighteq w) \to_M^* (h, \triangleright z\underline{\#})$$

In maniera del tutto analoga si definisce la nozione di *WHILE*-calcolabilità, in cui usiamo la variabile $x$ per memorizzare sia il valore di ingresso che quello di uscita dei programmi.

**Definizione 1.4.6.** Un comando $C$ *calcola* $f : \mathbb{N} \to \mathbb{N}$, oppure $f$ è *WHILE-calcolabile*, se e solamente se $\forall n \in \mathbb{N}, \forall \sigma \in \mathrm{Var} \to \mathbb{N}$ tale che $\sigma(x) = n$

$$f(n) = m \text{ se e solamente se } \langle C, \sigma \rangle \to^* \sigma' \text{ e } \sigma'(x) = m$$

Finora abbiamo considerato funzioni da stringhe a stringhe o da naturali a naturali. Inoltre sia le MdT viste che i programmi *WHILE* e *FOR* hanno un solo dato di ingresso. Allora ci poniamo la seguente

*Domanda* 1.4.7. Ma se la $f$ fosse una funzione che opera su dati che non sono stringhe o memorie o numeri naturali? e se ha molti argomenti?

Si supera la difficoltà per mezzo di opportune *codifiche* dei dati, ovvero funzioni biunivoche che siano *effettive e facili*,[11] nel modo seguente

i) Dato $x$ in formato $A$ codificalo come $y$ in formato $B$.

ii) Applica la MdT a $y$ e se e quando ottieni $z$ in formato $B$,

iii) traduci $z$ dal formato $B$ al formato $A$.

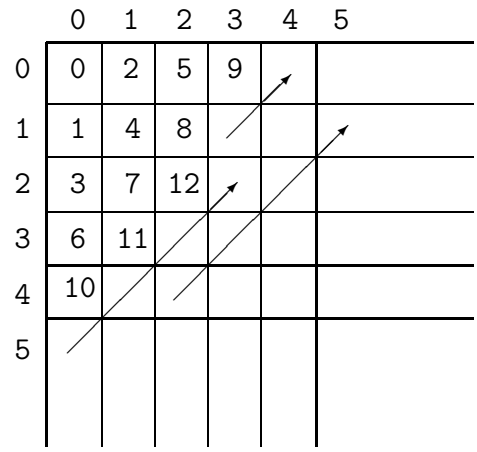
A titolo esemplificativo, vediamo adesso una codifica "classica."

**Esempio 1.4.8** (Coda di colomba)**.** La seguente funzione codifica coppie di naturali come un singolo naturale

$$(x, y) \mapsto \frac{1}{2}(x^2 + 2xy + y^2 + 3x + y)$$

ed è graficamente rappresentata come segue:

[11] Preciseremo in seguito cosa intendiamo per *effettivo* e per *facile*. Lo faremo considerando solo i numeri naturali, loro coppie ecc., quindi qui imbrogliamo un bel po', dando per buoni gli studi sull'argomento che si trovano in letteratura, e nella speranza di essere perdonati — tanto l'esame lo fate voi!




Risparmiamo al lettore sia la dimostrazione che questa funzione è biunivoca, sia lo sforzo per immaginare la funzione inversa, cioè la funzione di decodifica, che risulta essere definita cosí (si noti che nella colonna 0, l'elemento $n$-esimo è la somma dei primi $n$ naturali a partire da 1):

$$n \mapsto (n - \frac{1}{2}k \times (k+1), k - (n - \frac{1}{2}k \times (k+1))$$

dove $k = \lfloor \frac{1}{2}(\sqrt{1 + 8 \times n} - 1) \rfloor$.

*Domanda* 1.4.9. Esistono modelli di calcolo con dati più ricchi?

Certamente! ogni linguaggio di programmazione lo è (ma *NON lo sono le sue implementazioni.* Perché?). Ovviamente la presenza di dati con struttura piú ricca permette di migliorare la comprensibilità della rappresentazione delle funzioni — in termini informatici, la concisione e la leggibilità dei programmi, la loro modificabilità e molte altre proprietà interessanti dal punto di vista della produzione e dell'uso del software. Ma allora bisogna subito chiedersi se le codifiche mutino la natura dei problemi risolvibili e delle proprietà delle funzioni che consideriamo come calcolabili. Ovvero

*Domanda* 1.4.10. Le proprietà basilari dei formalismi e della classe delle funzioni calcolate cambiano al variare del formato dei dati su cui esse operano?

**NO**. I formalismi in questione sono tutti equivalenti se sono sufficientemente potenti, cioè se sono in grado di esprimere *tutte* le funzioni calcolabili (per esempio, le implementazioni dei linguaggi di programmazione *non* lo fanno, v. sopra). Le codifiche, proprio perché sono trasformazioni "facili" di

funzioni su certi dati in funzioni "equivalenti" su dati diversi, non allargano né restringono tale classe — bisognerebbe dimostrarlo, però! e forse far vedere qualche altro esempio di codifica, quale la rappresentazione di alberi come numeri, ricordando magari che una filza MP3 non è altro che un numero che codifica musica ...

Ci sentiamo allora liberi di considerare da qui in avanti solo funzioni con un solo argomento e i *numeri naturali* come nostri dati, prendendoci senza alcun timore il rischio di banalizzare il trattamento.

Vediamo adesso un esempio di funzione un po' strana, nella cui definizione si menziona la congettura di Goldbach, definita piú sotto, che l'autore sottopose nel 1742 in una lettera all'attenzione di Eulero, il quale non rispose mai; tuttora non si sa se la congettura sia vera o meno e per il momento essa è stata verificata "solo" sui numeri fino a 400 miliardi. Ritorneremo su questa funzione per esaminare le relazioni che intercorrono tra funzioni e algoritmi.

**Esempio 1.4.11.** $gb : \mathbb{N} \to \mathbb{N}$

$$gb(n) = \begin{cases} 0 & \text{se la congettura di Goldbach è vera} \\ 1 & \text{altrimenti} \end{cases}$$

**Definizione 1.4.12** (Congettura di Goldbach)**.**
$\forall m > 1$ si ha $2m = p_1 + p_2$ con $p_1, p_2$ primi.

Eccovi alcuni esempi di scomposizione à la Goldbach:

$$\begin{aligned} 22 &= 17 + 5 = 11 + 11 \\ 24 &= 17 + 7 \\ 36 &= 17 + 19 \end{aligned}$$

Abbiamo gli algoritmi, sia pure sotto forma di macchine di Turing o di programmi *WHILE* e abbiamo le funzioni (non necessariamente T- o *WHILE*-calcolabili). Allora siamo arrivati a porci una domanda cruciale:

*Domanda* 1.4.13. Qual è il rapporto tra funzioni e algoritmi?

Una funzione $f$ è un insieme di coppie, cioè $f$ associa l'argomento con il risultato (es. $doppio(n) = n + n$) senza dire *come fare a calcolarlo*. Di conseguenza, *non ci sono due funzioni* diverse *che per ogni argomento restituiscono lo stesso risultato*, il che riscritto in termini insiemistici si legge: *non esistono due insiemi* diversi *che hanno gli stessi elementi*! Per esempio, consideriamo ancora la funzione *doppio*: esiste uno e un solo insieme $\{(0,0),(1,2),(2,4),\dots\}$.

Un algoritmo (quando c'è!) invece specifica *come si calcola* il risultato a partire dall'argomento. In altre parole un algoritmo *calcola* o *rappresenta*, in

modo *finito*, una funzione. È facile vedere che ci possono essere più algoritmi che calcolano la stessa funzione — saremo più precisi in seguito: qui basta considerare un programma *WHILE* e aggiungergli un po' di `skip`.

Infine, riprendiamo la funzione $gb(n)$ dell'esempio 1.4.11. Non sappiamo a tutt'oggi se la congettura di Goldbach vale. Eppure un algoritmo per calcolarla esiste! ma non sappiamo quale esso sia, o meglio essi siano: quelli che danno come risultato 0 per tutti gli ingressi, o quelli che danno sempre come risultato 1? [12]

Finalmente siamo arrivati al nocciolo:

- Quali sono le funzioni calcolabili? Per ora sappiamo quali sono le Turing-calcolabili e le *WHILE*-calcolabili
- Di quali proprietà godono?
- Esistono funzioni (totali/parziali), ovvero problemi, che non sono calcolabili? ovvero per cui si *dimostra* che non esiste algoritmo che le calcoli? Tra questi problemi non calcolabili, ce n'è di interessanti? (Si noti che la funzione indefinita per ogni ingresso è (T- e *WHILE*-)calcolabile.)

Prima di affrontare queste domande, vedremo brevemente alcune differenze tra le MdT e i programmi *WHILE* e introdurremo un ulteriore formalismo.

Ricordiamo inoltre che in questa prima parte del corso studieremo algoritmi e funzioni, con maggior attenzione alle seconde. In termini classici, l'enfasi sarà sugli aspetti *estensionali*, perché ci occuperemo di ciò che è rappresentato (la semantica, il significato, la funzione) piuttosto di ciò che rappresenta (l'algoritmo, il programma): *cosa si calcola*, piuttosto che come si calcola.

La terza parte del corso terrà invece in maggior considerazione gli algoritmi: *come si calcola*.

[12] Gli intuizionisti rifiutano questo discorso: una entità matematica esiste se e solamente se la puoi costruire, e qui non esisterebbe (ancora)! Ciò che viene escluso è il *tertium non datur* aristotelico.

## 1.5 Due approcci alla calcolabilità

Abbiamo visto due modelli, i quali sottolineano due punti di vista leggermente diversi e complementari per descrivere algoritmi. In entrambi gli approcci, una configurazione è la coppia

(istruzione corrente, stato della memoria)

mentre cambia il modo con cui si interpretano le istruzioni.

Nel primo punto di vista, che per semplicità chiameremo approccio *hardware*, un algoritmo è una macchina in cui *l'insieme delle istruzioni rappresenta l'architettura*. Il fatto che il programma stesso sia considerato come una macchina può sorprendere e richiede quindi una spiegazione ulteriore, prendendo in esame le macchine di Turing. In effetti, la funzione di transizione rappresenta un programma e l'hardware pare essere composto solo dal nastro e dalla testina, dotata di un meccanismo di spostamento. Tuttavia, in una macchina di Turing si cabla l'intero programma: ogni macchina ha un numero fissato di stati e di simboli ed è completamente definita dalla sua funzione di transizione. Infatti non appena cambiamo quest'ultima, definiamo una nuova macchina. In altre parole, una macchina di Turing è la realizzazione di un *unico* algoritmo: l'unica cosa che varia è il dato di ingresso.

Analogamente, ogni altro modello nell'approccio hardware ha una memoria che è infinita tout court o è indefinitamente espandibile, e ogni suo elemento realizza un singolo algoritmo. Piú in generale, può addirittura accadere che si usino macchine via via piú grandi al crescere delle dimensioni del problema, che però sono tra loro simili o uniformi (si veda la cosidetta "circuit complexity"); vedremo un esempio di quanto detto subito prima della dimostrazione del teorema di Cook, in particolare quando useremo quella che vien detta tabella di computazione di una macchina di Turing (vedi pagina 184).

L'approccio software vede invece un algoritmo come un programma "interpretato" da un agente di calcolo, esso stesso una macchina (vedi il teorema di enumerazione 1.9.8), che può essere realizzata in:

- hardware (o firmware): la macchina fisica su cui "gira";
- software, cioè un programma a più basso livello: la macchina *astratta* che lo "esegue";

Anche in questo caso, la configurazione della macchina è, come prima, formata dalla coppia costituita da un puntatore all'istruzione corrente e dallo stato della memoria. A differenza dell'approccio hardware, qui i programmi

*non* sono parte integrante della macchina, ma sono contenuti nella memoria (cf. ancora il teorema 1.9.8). Di conseguenza, programmi più lunghi *non* richiedono agenti di calcolo di dimensioni maggiori: l'agente di calcolo o l'interprete è fisso e non cambiano solo i dati di ingresso, ma anche i programmi.

Tra i formalismi visti, i programmi *WHILE* hanno proprio quest'ultima caratteristica: l'interprete è (una qualunque realizzazione del)la semantica del linguaggio, la memoria è infinita, modellata attraverso la funzione $\sigma$, il contatore istruzioni è rappresentato dal programma che dobbiamo ancora eseguire (quest'ultimo punto è particolarmente evidente nell'approccio SOS alla semantica operazionale).

Introdurremo tra poco un terzo formalismo, da ascrivere tra quelli software: quello delle funzioni ricorsive, il quale è molto rilevante sia in informatica che nella teoria della calcolabilità classica, tanto che a volte questa è chiamata teoria delle *funzioni ricorsive*. Uso e ragione per l'introduzione dei tre principali formalismi che vedremo sono elencati qui sotto.

i) Funzioni Ricorsive *CHIAREZZA*
Sono la base per: Programmazione Funzionale; Semantica Denotazionale.

ii) Programmi *WHILE/FOR* *FAMILIARITÀ*
Sono la base per: Programmazione Imperativa; Semantica Operazionale; Complessità (Loop Programs).

iii) Macchine di Turing *SEMPLICITÀ*
Sono la base per: Descrizione Macchine; Algoritmi; Complessità.

Come già detto, i primi due paradigmi si possono classificare nell'approccio software, mentre il terzo è un paradigma più vicino all'hardware. Questa distinzione è largamente arbitraria e formalmente vaga, ma può spiegare perché vi siano differenti presentazioni della materia; inoltre può apparire sfuggente a questo punto del corso, ma verrà auspicabilmente chiarita nella sua terza parte, ad esempio discutendo il già citato teorema di Cook.

Ovviamente tutta la trattazione che segue potrebbe essere basata su uno qualunque di questi formalismi, e su altri ancora che non abbiamo menzionato, tra cui ad esempio le grammatiche e sistemi di Post, gli algoritmi di Markov, il $\lambda$-calcolo, le macchine a registri (che vedremo di sfuggita), e molti altri ancora. Ciascuno di essi chiarisce in maggiore o minore misura i vari aspetti che andremo a considerare nel seguito senza per altro definire classi diverse di funzioni calcolabili o classi di complessità, come abbiamo già piú volte ripetuto.

## 1.6 Funzioni ricorsive primitive

Cominciamo a vedere un paio di esempi di funzioni definite per ricorrenza che sono molto popolari. In seguito introdurremo la classe delle funzioni *ricorsive primitive*, della cui debolezza e del modo di sopperirvi si discuterà nel prossimo capitolo.

La prima funzione che consideriamo è il fattoriale, definita come spesso si fa, da una coppia di equazioni, una nel caso (base) in cui l'argomento sia 0, l'altra nel caso (ricorsivo) in cui l'argomento non sia nullo.

$$\textit{Fattoriale} \qquad \begin{cases} !(0) & = 1 \\ !(x+1) & = (x+1) \times !(x) \end{cases}$$

Eccone una versione in *WHILE*, che "lascia" il valore di $!(x)$ nella variabile $fatt$ (il lettore è invitato a scriverne una versione in *FOR*).

$$\begin{array}{l} fatt \ := \ 1; \\ \texttt{while}\ 0 < x\ \texttt{do} \\ \qquad\qquad\qquad fatt \ := \ fatt \times x; \\ \qquad\qquad\qquad x \ := \ x - 1 \end{array}$$

La seconda funzione, ben nota agli informatici (e ai matematici, botanici, fisici, architetti, ecc.) è la funzione di Fibonacci, la cui definizione data qui sotto ha, per ovvie ragioni, due casi base

$$\textit{Fibonacci} \qquad \begin{cases} fib(0) & = 0 \\ fib(1) & = 1 \\ fib(x+2) & = fib(x+1) + fib(x) \end{cases}$$

La funzione di Fibonacci si può anche scrivere in forma chiusa, non ricorsiva:

$$fib(x) = \frac{\sqrt{5}}{5}(\frac{1+\sqrt{5}}{2})^x - \frac{\sqrt{5}}{5}(\frac{1-\sqrt{5}}{2})^x.$$

Adesso proviamo a formalizzare i modi diversi che abbiamo usato per definire le funzioni *Fattoriale* e *Fibonacci*. Nel far ciò, usiamo quella che si chiama $\lambda$-notazione per individuare all'interno di una espressione (scritta seguendo un'opportuna sintassi) che descrive una funzione quali sono i suoi argomenti. Scriveremo quindi

$$\lambda x, y, z.espressione$$

quando gli argomenti della *espressione* sono proprio $x$, $y$ e $z$; si dice anche che $x$, $y$ e $z$ appaiono *legate* da $\lambda$ in *espressione*. Invece, un qualsiasi altro

simbolo (di variabile) $w$ che appaia in *espressione*, *non* sarà da considerarsi un argomento della funzione rappresentata da *espressione*; a volte un tale $w$ viene chiamato *libero* in *espressione*.[13] Ad esempio,

$$\lambda x, y.\, x + y \qquad \text{(o piú precisamente } \lambda x.\lambda y.\, x + y)$$

è ovviamente la funzione che somma $x$ ad $y$, una volta che si sia interpretato il simbolo + come si suole. In questo caso potremmo anche scrivere

$$somma(x, y) = x + y$$

dando cosí il nome *somma* alla funzione, magari per usi futuri (si confrontino questi due modi di definire funzioni con nome o senza nome con i modi di definire e costruire funzioni, in voga nei linguaggi funzionali, tipo ML).

Introduciamo adesso la prima classe di funzioni che ci interessano. Per seguire l'uso corrente e per non appesantire eccessivamente la notazione, di seguito mescoleremo la sintassi e la semantica, dicendo che definiamo una funzione $f$ dai naturali ai naturali, mentre in realtà stiamo definendo un *algoritmo* che *rappresenta* la "vera" funzione $f$.

**Definizione 1.6.1.** La classe delle funzioni *ricorsive primitive* è la minima classe di funzioni $\mathcal{C}$ da $\mathbb{N}^n$, $n \geq 0$, in $\mathbb{N}$ cui appartengono:

I *Zero* $\lambda x_1, \ldots, x_k.\, 0$ con $k \geq 0$

II *Successore* $\lambda x.\, x + 1$

III *Identità (o proiezioni)* $\lambda x_1, \ldots, x_k.\, x_i$ con $1 \leq i \leq k$

dette anche *schemi primitivi di base*, e che è chiusa per:

IV *Composizione*
Se $g_1, \ldots, g_k \in \mathcal{C}$ sono funzioni in $m$ variabili, ed $h \in \mathcal{C}$ è una funzione in $k$ variabili, appartiene a $\mathcal{C}$ anche la loro composizione

$$\lambda x_1, \ldots, x_m.\, h(g_1(x_1, \ldots, x_m), \ldots, g_k(x_1, \ldots, x_m))$$

V *Ricorsione Primitiva*
Se $h \in \mathcal{C}$ è una funzione in $k + 1$ variabili, $g \in \mathcal{C}$ è una funzione in $k - 1$ variabili, allora appartiene a $\mathcal{C}$ anche la funzione $f$ in $k$ variabili definita da:

$$\begin{cases} f(0, x_2, \ldots, x_k) & = & g(x_2, \ldots, x_k) \\ f(x_1 + 1, x_2, \ldots, x_k) & = & h(x_1, f(x_1, x_2, \ldots, x_k), x_2, \ldots, x_k) \end{cases}$$

[13] Piú precisamente si dovrebbe parlare di apparizioni, o con un calco dall'inglese, *occorrenze* legate o libere di quelle o questa variabile. Nell'ultimo caso, la funzione descritta da *espressione* è da interpretarsi come una funzione *parametrica* in $w$, nel senso dato alla parola nei corsi di Analisi.

Si noti che, se $k = 1$, allora $g$ ha 0 argomenti ed è ricorsiva primitiva in quanto è una costante. Così come avviene per le MdT, anche la definizione della classe delle funzioni ricorsive primitive ha moltissime varianti, tutte equivalenti tra loro, nel senso che le classi che esse definiscono coincidono tutte con la classe $\mathcal{C}$. Per esempio, invece di avere lo schema I si può avere lo schema che introduce le costanti:

$$\text{I}' \ \textit{Costanti} \ \lambda x_1, \ldots, x_k.\ m \text{ con } k, m \geq 0.$$

Chiaramente, componendo un numero opportuno di volte la funzione successore a partire dalla funzione costante 0 si ottengono tutte le costanti $m$ introdotte in un sol colpo dallo schema appena definito (si veda sotto come si calcola). Altre varianti piuttosto comuni riguardano la posizione in cui appare la chiamata ricorsiva alla funzione $f$ nella parte destra della seconda equazione nello schema V (v. oltre).

Poichè $\mathcal{C}$ è la *minima* classe che soddisfa le condizioni espresse sopra, affinchè $f$ sia ricorsiva primitiva, occorre e basta che ci sia una successione finita, o *derivazione*, della seguente forma

$$f_1, f_2, \ldots, f_n$$

tale che $f = f_n$ e $\forall i$ tale che $1 \leq i \leq n$ vale uno dei due casi seguenti:

- $f_i \in \mathcal{C}$ per I–III (ovvero $f_i$ è definita secondo gli schemi di base) o
- $f_i$ è ottenibile mediante applicazione delle regole IV e V da $f_j,\ j < i$ (ovvero $f_i$ è definita da funzioni già definite precedentemente).

**Esempio 1.6.2.** Vogliamo definire $f_5 = \lambda x, y.\ x + y$. Una possibile derivazione è la seguente, in cui abbiamo indicato accanto ad ogni $f_i$ la regola applicata per ottenerla.

$$\begin{array}{ll}
f_1 = \lambda x.\ x & III \\
f_2 = \lambda x.\ x + 1 & II \\
f_3 = \lambda x_1, x_2, x_3.\ x_2 & III \\
f_4 = f_2(f_3(x_1, x_2, x_3)) & IV \\
\begin{cases} f_5(0, x_2) = f_1(x_2) \\ f_5(x_1 + 1, x_2) = f_4(x_1, f_5(x_1, x_2), x_2) \end{cases} & V
\end{array}$$

Vediamo adesso come si calcola $f_5(2, 3)$, cioè la somma di 2 e 3. Nel passare da un rigo al successivo, espandiamo la "chiamata di funzione" sottolineata (detta *redex*). Si noti che usiamo la regola di valutazione *interna sinistra*: il redex prescelto è quello piú interno all'espressione da valutare e piú a sinistra.

$$
\begin{aligned}
&\underline{f_5(2,3)} = \\
&\underline{f_4(1, f_5(1,3), 3)} = \\
&f_4(1, f_4(0, \underline{f_5(0,3)}, 3), 3) = \\
&f_4(1, f_4(0, \underline{f_1(3)}, 3), 3) = \\
&f_4(1, \underline{f_4(0,3,3)}, 3) = \\
&f_4(1, f_2(\underline{f_3(0,3,3)}), 3) = \\
&f_4(1, \underline{f_2(3)}, 3) = \\
&\underline{f_4(1,4,3)} = \\
&\underline{f_2(f_3(1,4,3))} = \\
&\underline{f_2(4)} = \\
&5
\end{aligned}
$$

Il lettore attento si sarà certamente reso conto che si potrebbe fare una scelta diversa dei redex, per esempio selezionando l'espressione piú esterna che può essere valutata; si noti tuttavia che per far questo bisogna poter legare a una variabili $x$ non solo un numero naturale, ma anche un'intera espressione. Questa regola di valutazione viene chiamata *esterna*; usiamola nell'esempio di sopra.

$$
\begin{aligned}
&\underline{f_5(2,3)} = \\
&\underline{f_4(1, f_5(1,3), 3)} = \\
&\underline{f_2(f_3(1, f_5(1,3), 3))} = \\
&\underline{f_3(1, f_5(1,3), 3)} + 1 = \\
&\underline{f_5(1,3)} + 1 = \\
&\underline{f_4(0, f_5(0,3), 3)} + 1 = \\
&\underline{f_2(f_3(0, f_5(0,3), 3))} + 1 = \\
&\underline{f_3(0, f_5(0,3), 3)} + 1 + 1 = \\
&\underline{f_5(0,3)} + 2 = \\
&\underline{f_1(3)} + 2 = \\
&3 + 2 = 5
\end{aligned}
$$

In realtà si può dimostrare che la scelta del redex non è determinante, ovvero che le due regole di valutazione sono equivalenti.[14] In questo caso, le riduzioni

[14]Questo non è piú vero in questi termini per la classe di funzioni che introdurremo tra due capitoli, in quanto la regola di valutazione esterna potrebbe portare a un risultato anche in presenza di un calcolo non terminante, eseguito usando la regola interna sinistra. In effetti, c'è un teorema che assicura che tutte le volte che, applicando la regola interna, si arriva a un risultato, si ottiene il medesimo risultato con la regola esterna; mentre il viceversa può non essere vero. Piú in generale, si può dire che tale risultato è raggiungibile anche *senza scegliere a priori alcuna regola* per selezionare il redex.

necessarie, sia con la regola interna sinistra che con quella destra sono 9, se non contiamo le applicazioni della funzione primitiva successore. In generale questo può non essere vero e per rendersene conto si consideri la seguente espressione

$$f_3(f_5(2,3),0,f_5(2,3))$$

la cui valutazione con la regola esterna richiede una sola riduzione e 19 con la regola interna. (È sempre vero che in questo formalismo la regola esterna è piú efficiente di quella interna, ovvero richiede meno o al piú lo stesso numero di passi di calcolo?)

Naturalmente per noi informatici è di grande importanza usare una regola di valutazione che minimizzi il numero di passi necessari al calcolo e anche lo spazio necessario a conservare i risultati intermedi – si pensi ad esempio alla realizzazione di interpreti o compilatori efficienti per linguaggi funzionali (e le regole di valutazione *by value, by need, lazy* ecc.).

Concludiamo l'esempio aggiungendo alla derivazione data sopra per la funzione di somma la seguente clausola, che ci permette di ottenere una definizione per la funzione che raddoppia il suo argomento $f_6 = \lambda x.\ 2 \times x$

$$f_6 = f_5(f_1(x), f_1(x)) \quad IV$$

Bene, abbiamo un bel formalismo, semplice, elegante, anche se piuttosto verboso. È facile vedere che le usuali funzioni ($x \times y$, $x^y$, $x \dot{-} y$, ecc.) sono tutte ricorsive primitive. Adesso rendiamoci la vita piú comoda usando identificatori $x, y, z, ...$ al posto di $x_1, x_2, ...$ (quanti simboli servono per rappresentare le $x_i$?) e permettendo la ricorsione in posizioni diverse dalla seconda. Definiamo prima le funzioni proiezione (a due posti) che restituiscono il primo e il secondo argomento e quella che restituisce il predecessore del suo argomento, se maggiore di 0, altrimenti restituisce 0 (e chiamiamola *pred*, dandogli un nome appena piú evocativo di quelli usati fin'ora e semplifichiamoci la vita stipulando che 0 è una funzione a nessun argomento):

$$f_7(x,y) = y \qquad f_8(x,y) = x \qquad \begin{cases} pred(0) & = & 0 \\ pred(x+1) & = & f_8(x, pred(x)) \end{cases}$$

Poi introduciamo la funzione ausiliaria che sottrae (limitatamente) 1 al secondo dei suoi tre argomenti per definire l'ultima funzione ausiliaria che sottrae (limitatamente) il primo argomento al secondo:

$$f_9(x,y,z) = pred(f_3(x,y,z)) \qquad \begin{cases} f_{10}(0,y) & = & f_1(y) \\ f_{10}(x+1,y) & = & f_9(x, f_{10}(x,y), y) \end{cases}$$

Finalmente ci siamo: il meno limitato è:

$$x \dot{-} y = f_{10}(f_7(x,y), f_8(x,y))$$

Il lettore avrà certamente notato un'ulteriore semplificazione: nell'ultima definizione abbiamo usato la notazione infissa che è assai piú comune. Semplifichiamoci ancora la vita: diamo per intese le necessarie composizioni e proiezioni e guardiamo alle seguenti definizioni, in cui riappare la definizione della somma, indicata da $+$ (da non confondersi con il successore $+1$), già vista in tutti i dettagli nell'esempio 1.6.2. Le altre due funzioni sono il prodotto e l'esponente, in cui stipuliamo che $0^0 = 1$ (in alternativa a tale ipotesi sventurata si potrebbe porre come case base $(x+1)^0 = 1$, ma allora quanto varrebbe $0^0 = 1$? la funzione che definiamo sarebbe indefinita sullo 0? forse però i nostro professori di matematica sarebbero meno scandalizzati.). Una rapida ispezione, e forse un paio di esempi, dovrebbero convincerci che il "+" generalizza il successore $+1$, nel senso che questo viene applicato $x$ volte a $y$, e che il $\times$ generalizza il $+$ e infine che l'esponente generalizza il $\times$ (ci sarà una funzione che generalizza l'esponente? si veda la nota a pagina 37).

$$\begin{cases} 0+y & = y \\ (x+1)+y & = (x+y)+1 \end{cases} \qquad \begin{cases} 0 \times y & = 0 \\ (x+1) \times y & = (x \times y)+y \end{cases} \qquad \begin{cases} x^0 & = 1 \\ x^{(y+1)} & = x \times (x^y) \end{cases}$$

È facile convincersi, magari ricorrendo ai testi di riferimento, per esempio [Lewis-Papadimitriou], che è possibile estendere lo schema di ricorsione primitiva permettendo la definizione per casi, e non solo di 0 e di $x+1$ diverso da 0 come fatto fin'ora.

Per scopi futuri, definiamo anche le relazioni ricorsive primitive.

**Definizione 1.6.3.** Diciamo che la relazione $P(x_1, \ldots, x_k) \subseteq \mathbb{N}^k$ è *ricorsiva primitiva* se lo è la sua funzione caratteristica $\chi_P$, definita come

$$\chi_P(x_1, \ldots, x_k) = \begin{cases} 1 & \text{se } (x_1, \ldots, x_k) \in P \\ 0 & \text{se } (x_1, \ldots, x_k) \notin P \end{cases}$$

Come esempio di relazione ricorsiva primitiva vediamo l'eguaglianza, definendo la sua funzione caratteristica $\chi_=$ (ricorrendo su due argomenti contemporaneamente):

$$\chi_=(0,0) = 1 \quad \chi_=(x+1, y+1) = \chi_=(x,y) \quad \chi_=(0, y+1) = 0 \quad \chi_=(x+1, 0) = 0$$

(Si noti come lo schema di ricorsione nella seconda clausola sia stato usato su entrambi gli argomenti: una definizione piú pedante richiederebbe di valutare sia $x \dot{-} y$ che $y \dot{-} x$ e restituire 1 quando entrambi restituiscono 0.)

Inoltre ci servirà sapere che

- $R = \{x \in \mathbb{N} \mid x \text{ è un numero primo}\}$ è ricorsiva primitiva
- (unica fattorizzazione) se $p_0 < \ldots < p_k \ldots$ sono i numeri primi, cioè $R = \{p_0, \ldots, p_k \ldots\}$, allora per ogni $x \in \mathbb{N}$ esiste un numero *finito* di esponenti $x_i \neq 0$ tali che $x = p_0^{x_0} p_1^{x_1} \ldots p_n^{x_n} \ldots$;
- la funzione $(x)_i$ che restituisce l'esponente dell'$i$-esimo fattore $p_i$ della fattorizzazione di $x$ è ricorsiva primitiva.

Una conseguenza assai interessante è che ogni sequenza di numeri naturali $n_0 n_1 \ldots n_k$ può essere codificata *univocamente* come $n = p_0^{n_0+1} p_1^{n_1+1} \ldots p_k^{n_k+1}$ (con $p_i$ primi), ovvero come il prodotto di un numero *finito* di fattori, e viceversa. In altre parole, data la sequenza $n_0 n_1 \ldots n_k$ esiste un unico $n$ che verifica l'uguaglianza $n = p_0^{n_0+1} p_1^{n_1+1} \ldots p_k^{n_k+1}$ e viceversa. Questa è la base per dimostrare che le *funzioni di codifica sono ricorsive primitive*.

Vediamo ora, intuitivamente e non in dettaglio, come si può sfruttare questo teorema per costuire una codifica delle macchine di Turing e delle loro computazioni. A essere precisi *non* faremo vedere una codifica, ma delineremo a spanne una funzione che è iniettiva, ma *non* surgettiva. Ritorneremo piú avanti su questo fatto, legato a pigrizia e desiderio di tenere le cose il piú semplici possibile. Sia

$$M = (Q, \Sigma, \delta, q_0)$$

una MdT con

$$Q = \{q_0, \ldots, q_n\} \text{ e } \Sigma = \{\sigma_0, \ldots, \sigma_m\}$$

Ogni quintupla $(q_i, \sigma_j, q_k, \sigma_\ell, D) \in \delta$ è codificata come

$$p_0^{i+1} \times p_1^{j+1} \times p_2^{k+1} \times p_3^{\ell+1} \times p_4^{m_D}$$

dove $p_0 < \ldots < p_4$ sono numeri primi. Inoltre, consideriamo i simboli di direzione $D$ come simboli veri e propri: $L = \sigma_{m+2}$, $R = \sigma_{m+3}$, $- = \sigma_{m+4}$ e poniamo allora $m_D \in \{m_L = m + 2, m_R = m + 3, m_- = m + 4\}$. Infine, facciamo in modo che lo stato "speciale" $h$ sia identificato come lo stato $q_{n+1}$.

Grazie al teorema di unica fattorizzazione, a ogni quintupla è associato un *solo* intero e viceversa; questa osservazione è la base per dimostrare che la funzione che stiamo proponendo è iniettiva.

Ora ordiniamo l'insieme di quintuple che caratterizza $M$, ponendo ad esempio $(q_i, \sigma_j, q_k, \sigma_\ell, D)$ minore di $(q_{i'}, \sigma_{j'}, q_{k'}, \sigma_{\ell'}, D')$ se $i < i'$ o se $i = i'$ e $j < j'$, e cosí via. A questo punto, abbiamo una successione di quintuple e ciascuna di esse viene codificata come un numero, ottenendo una successione $a_0 a_1 \ldots a_n$ di numeri naturali, tali che $a_i \neq a_j$ se $i \neq j$. Adesso anche questa

successione viene codificata, e si ottiene finalmente il numero $i$, detto anche *indice*, che vogliamo associare alla macchina $M$.

Come già accennato sopra, è immediato rendersi conto che la funzione proposta non è surgettiva: non sempre da un numero $i$ posso recuperare una macchina $M$; ma il problema si aggira facilmente. Infatti, uno decodifica l'indice $i$; se il risultato ottenuto è una MdT, bene; altrimenti $i$ non è nell'immagine (ovvero non è un vero e proprio indice) e butto via tutto, oppure restituisco un valore speciale. In altre parole, ci si riduce a considerare la coppia codifica/decodifica come operanti rispettivamente dalle MdT a $\mathbb{N}$ e dalla *immagine* della codifica alle MdT (oppure da $\mathbb{N}$ alle MdT piú il valore speciale di cui sopra). Si può certamente far meglio e avere una vera funzione di codifica che sia quindi biunivoca, al prezzo di una definizione molto astuta (e complicata): l'ha già fatto per noi Kurt Gödel, seppure non per le MdT, tanto che il procedimento sopra delineato viene chiamato *gödelizzazione* e *numero di Gödel* di $M$ il numero cosí ottenuto dalla macchina $M$ data.

Basti qui notare che abbiamo un modo per *enumerare* le macchine di Turing, ovvero di metterle in lista basandoci *unicamente* sui simboli usati per definir(e le loro quintup)le; inoltre il modo è "facile", perché le funzioni ricorsive primitive lo sono, in un senso che sarà piú chiaro nel seguito.

Naturalmente il procedimento accennato sopra può essere applicato alle configurazioni e poi anche alle computazioni, che non sono altro che successioni di configurazioni. Un numero naturale allora può essere interpretato come la *codifica di un'intera computazione!* Del resto, un programma caricato in memoria può esser visto come una successione di bit, la quale rappresenta un numero naturale e cosí la sequenza di configurazioni attraversate.

Ovviamente il medesimo procedimento di enumerazione può essere applicato anche ai programmi FOR e WHILE, alle funzioni ricorsive primitive e a quelle $\mu$-ricorsive che introdurremo nel prossimo capitolo, nonché a ogni formalismo per rappresentare funzioni che rispetti i vincoli posti nel capitolo 1.1. Quindi la codifica che ci ha regalato Gödel è davvero un'arma potentissima, che permette di enumerare, o se volete di digitalizzare le rappresentazioni delle funzioni e il loro calcolo (e anche di altre realtà, per esempio immagini o musica!).

Ritorniamo ora alle nostre funzioni definite per ricorsione primitiva. È facile verificare con un ragionamento induttivo che tutte le funzioni definibili mediante gli schemi di ricorsione primitiva sono *totali*.

Dal punto di vista informatico è interessante notare che le funzioni ricorsive primitive e il linguaggio *FOR* sono equivalenti, nel senso che data una funzione ricorsiva primitiva si può scrivere un programma *FOR* che, con gli stessi argomenti, produce lo stesso risultato e viceversa.

*Domanda* 1.6.4. Abbiamo trovato il formalismo giusto, che esprime tutte le funzioni calcolabili? e per di piú solo quelle totali? Purtroppo **NO**

Infatti esiste la funzione di Ackermann, che *non è definibile* mediante gli schemi di ricorsione primitiva I-V, che è totale e ha una definizione che intuitivamente è accettabilissima. Vediamola, osservando che una sola delle regole seguenti è applicabile una volta fissati gli argomenti $x, y, z$ [15]

$$\begin{aligned}
A(0,0,y) &= y \\
A(0,x+1,y) &= A(0,x,y)+1 \\
A(1,0,y) &= 0 \\
A(z+2,0,y) &= 1 \\
A(z+1,x+1,y) &= A(z,A(z+1,x,y),y).
\end{aligned}$$

**Nota**: C'è una "doppia" ricorsione, ma tutti i valori su cui si ricorre decrescono, e quindi i valori di $A(z,x,y)$ sono definiti in termini di un numero *finito* di valori della funzione $A$ applicata ad argomenti $z'$ e $x'$ tali che $z' \leq z$ e $x' < x$. Quindi intuitivamente $A$ è calcolabile.

Ma cosa calcola $A$? Una sorta d'esponenziale generalizzato (si tenga conto che il $+$ è la funzione *successore* generalizzata, il $\times$ è il $+$ generalizzato, la funzione esponente è il $\times$ generalizzato). [16] Infatti:

$$\begin{aligned}
A(0,x,y) &= y+x \\
A(1,x,y) &= y \times x \\
A(2,x,y) &= y^x \\
A(3,x,y) &= \underbrace{y^{\cdot^{\cdot^{y}}}}_{x \text{ volte}}
\end{aligned}$$

Piú in dettaglio, le prime due clausole definiscono la somma di $x$ e $y$, la terza e la quarta sono la base per la quinta. Per vedere cosa fa la quinta, si consideri la seguente successione

[15] C'è una variante piú semplice di questa funzione in cui si può eliminare la variabile $y$, dopo averla considerata identicamente uguale a 1:

$$\begin{aligned}
A'(0,x) &= x+1 \\
A'(z+1,0) &= A'(z,1) \\
A'(z+1,x+1) &= A'(z,A'(z+1,x))
\end{aligned}$$

[16] L'esempio 1.6.2 e le definizioni a pagina 34 mostrano che la somma $x+y$ è il *successore* applicato $x$ volte ad $y$; che il prodotto $x \times y$ si ottiene sommando $x$ volte $y$; e che infine l'esponente $x^y$ è il prodotto iterato.

$$w_0 = A(z+1,0,y) = \begin{cases} 0 & \text{se } z+1 = 1 \ (3^\circ \text{ equazione}) \\ 1 & \text{se } z+1 > 1 \ (4^\circ \text{ equazione}) \end{cases}$$

$$\begin{aligned} w_1 = A(z+1,1,y) &= A(z, w_0, y) \\ w_2 = A(z+1,2,y) &= A(z, w_1, y) \\ &\vdots \\ w_x = A(z+1,x,y) &= A(z, w_{x-1}, y) \end{aligned}$$

Sfortunatamente, la funzione di Ackermann cresce piú velocemente di ogni funzione ricorsiva primitiva. Per cui vale il seguente teorema, che non dimostriamo — ahimé e buon per voi! [v. Bernasconi-Codenotti]; intuitivamente, la dimostrazione si basa sul fatto che questa funzione mette in piedi un numero di chiamate a sé stessa maggiori di quante ne possa fare qualsiasi funzione ricorsiva primitiva.

**Teorema 1.6.5.** *La funzione di Ackermann non è ricorsiva primitiva.*

Quindi abbiamo trovato una funzione *totale*, interessante, che non è ricorsiva primitiva. A margine, la funzione di Ackermann caratterizza anche una classe di problemi combinatorî, o meglio una classe di complessità (si veda la III parte delle note).

Evidentemente tra le funzioni che si riescono a definire secondo gli schemi di ricorsione primitiva I–V (e che quindi sono calcolabili *e* totali) ne manca almeno una. Di piú e peggio: ne mancano moltissime, ad esempio tutte quelle che si ottengono dalla funzione di Ackermann sommandogli una costante. Ma non si potrebbe aggiungere a forza la funzione di Ackermann tra le ricorsive primitive, o meglio estendere i suoi schemi con quella doppia ricorsione e ottenere un formalismo che esprime *solo* le funzioni totali e le esprime *tutte*? La questione diventa allora la seguente.

*Domanda* 1.6.6. Esiste un formalismo capace di esprimere *tutte e sole* le funzioni totali? Sfortunatamente, o meglio fortunatissimamente **NO**.

Per vederlo, definiremo nel capitolo seguente una tecnica di dimostrazione, detta *diagonalizzazione* e la useremo nel caso delle funzioni ricorsive primitive. Deve essere ben chiaro però che questo metodo si applica a *qualsiasi* formalismo che esprime *solo* funzioni totali.

## 1.7 Diagonalizzazione

Vediamo adesso una tecnica basilare della teoria della calcolabilità, che va sotto il nome di *diagonalizzazione*. Essa è strettamente legata alla dimostrazione che Cantor diede della non numerabilità dell'insieme dei sottinsiemi dei numeri naturali. Il modo con cui applichiamo la diagonalizzazione alle funzioni ricorsive primitive è in realtà indipendente da questo formalismo, essendo invece del tutto generale: si applica infatti a *tutti* i formalismi con cui si possano definire *solo* funzioni totali. Di seguito usiamo questa tecnica per dimostrare che le funzioni ricorsive primitive non sono tutte le funzioni calcolabili totali in quanto ne manca almeno una che intuitivamente lo è. Inoltre, la stessa dimostrazione fa vedere che comunque si estenda la classe delle funzioni ricorsive primitive a contenere *solo* funzioni totali, si ricasca nello stesso problema: si può costruire una funzione intuitivamente calcolabile non esprimibile con quella estensione. Di conseguenza, la classe delle funzioni ricorsive primitive, o qualunque classe che la estenda con *solo* funzioni totali calcolabili, non conterrà *tutte* le funzioni intuitivamente calcolabili. Quella che segue non è una vera dimostrazione, ma solo una traccia di come si dovrebbe procedere.

i) Ogni derivazione di una funzione ricorsiva primitiva è una stringa *finita* di simboli presi da un alfabeto *finito*. Quindi tali rappresentazioni si possono enumerare, per esempio con la funzione di Gödel, e indichiamo con $f_n$ la funzione definita dalla $n$-esima derivazione.

ii) Definisci $g(x) = f_x(x) + 1$. Questa è effettivamente calcolabile: prendi la $x$-esima definizione, applicala avendo come argomento il suo stesso indice $x$, trova il risultato e sommagli 1. (Si noti che la scelta dell'algoritmo dipende dall'argomento.) Inoltre, è facile vedere che la funzione $g$ è totale.

iii) La $g$ non si trova nella lista delle funzioni ricorsive primitive, perché $\forall n.\ g(n) \neq f_n(n)$, e quindi $\forall n.\ g \neq f_n$, o meglio la funzione calcolata dalla $g$ su $n$ restituisce un valore diverso dalla funzione calcolata dalla $f_n$ su $n$.

Come già anticipato, l'argomento usato sopra per le funzioni ricorsive primitive si applica ad *ogni* formalismo che definisca *solo* funzioni totali: basta costruire l'elenco delle funzioni definite in quel formalismo come fatto nel passo (i) di sopra, enumerandole come fatto nel capitolo precedente per le MdT, e poi diagonalizzare come nel passo (ii): la funzione cosí definita non appare nell'elenco costruito.

Quindi siamo obbligati a considerare anche *funzioni parziali*, che indicheremo spesso mediante lettere dell'alfabeto greco, quali $\varphi$ e $\psi$. Per fortuna, la diagonalizzazione non si applica alle funzioni parziali. Infatti sia $\psi_n$ la funzione con $n$-esima definizione, cioè che appare in posizione $n$-esima nella lista (per esempio calcolata dall $n$-esima MdT), e proviamo a diagonalizzare. Definiamo quindi

$$\varphi(x) = \psi_x(x) + 1$$

Supponiamo adesso che $\varphi$ sia rappresentata dall'$n$-esimo algoritmo: non posso tuttavia concludere $\varphi \neq \psi_n$ perché $\psi_n(n)$ può non essere definita!

Ma se prendessi proprio quegli algoritmi che definiscono funzioni totali per applicare la diagonalizzazione? In questo modo otterrei nuovamente una funzione che non trovo nella lista. Vedremo alla fine di questa parte del corso che questa cosa non si può fare effettivamente, cioè che *non esiste* un algoritmo che permetta di scegliere nella lista proprio le definizioni di funzioni totali.

Inoltre, fortunatamente le funzioni parziali hanno senso. Per esempio, si consideri la funzione seguente, che è definita solo se $y \neq 0$

$$div(x, y) = \lfloor x/y \rfloor.$$

C'è ancora un sospetto da fugare. Si potrebbero estendere tutte le funzioni parziali a funzioni totali, come facciamo nell'esempio seguente rendendo totale la funzione $div$ alla funzione $div^*$ come suggerito piú avanti? La risposta è **NO**, perché, come vedremo alla fine della prima parte del corso, *non sempre* c'è un algoritmo che calcola la funzione estesa (nel nostro caso $div^*$).

**Esempio 1.7.1.** Il primo modo è quello di definire accuratamente il dominio della funzione, per esempio ponendo

$$div : \mathbb{N} \times \mathbb{N}\backslash\{0\} \to \mathbb{N}$$

Però la funzione risultante non copre tutte le coppie di naturali. Possiamo far meglio: posto $* \notin \mathbb{N}$, abbiamo ancora una funzione su tutte le coppie di naturali [17]

$$div^* : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \cup \{*\}$$
$$div^*(x, y) = \begin{cases} div(x, y) & \text{se } y \neq 0 \\ * & \text{se } y = 0 \end{cases}$$

Poiché non possiamo e nemmeno vogliamo liberarci della parzialità, bisogna trovare il modo per poter definire anche funzioni parziali.

[17] Il suo codominio non sono però i naturali; se volessi essere pignolo allora potrei definire $div^*(x, 0) = 0$, facendo inorridire i nostri professori di matematica delle superiori.

## 1.8 Funzioni ricorsive generali

Arricchiamo di seguito gli schemi per la definizione delle funzioni ricorsive primitive con un nuovo schema, attraverso il quale è possibile esprimere anche funzioni parziali. In esso, si fa uso dell'operatore $\mu$, detto di *minimizzazione*, il quale applicato a un insieme di numeri naturali ne restituisce il minimo (se c'è! ovvero se l'insieme in questione non è vuoto).

**Definizione 1.8.1** (funzioni $\mu$-ricorsive)**.** La classe delle *funzioni $\mu$-ricorsive* (o *ricorsive generali*) è la minima classe $\mathcal{R}$ tale che soddisfa le condizioni

I–V per le ricorsive primitive e

VI. (Minimizzazione). Se $\varphi(x_1, \ldots, x_n, y) \in \mathcal{R}$ in $n+1$ variabili, allora la funzione $\psi$ in $n$ variabili è in $\mathcal{R}$ se è definita da

$$\psi(x_1, \ldots, x_n) = \mu y[\varphi(x_1, \ldots, x_n, y) = 0 \text{ e } \forall z \leq y.\ \varphi(x_1, \ldots, x_n, z) \downarrow] \qquad (*)$$

A volte, le funzioni $\mu$-ricorsive *totali* le chiameremo semplicemente *ricorsive*, soprattutto per ragioni storiche. Nota bene: *non* sono *solo* le funzioni ricorsive primitive! (in cui, per esempio manca la funzione di Ackermann che è ricorsiva ma non ricorsiva primitiva.)

Una funzione $\mu$-ricorsiva è intuitivamente calcolabile? **Sí**: l'algoritmo "intuitivo" che la calcola è composto da un ciclo in cui si incrementa la variabile $y$ (inizialmente posta a 0), si calcola la $\varphi$ e si ripetono questi passi finché il risultato non è 0. I primi passi dell'esecuzione di questo algoritmo potrebbero essere dunque:

1. calcola $\varphi(x_1, \ldots, x_n, 0)$; se il risultato è 0 allora $\psi(x_1, \ldots, x_n) = 0$;
2. altrimenti calcola $\varphi(x_1, \ldots, x_n, 1)$; se il risultato è 0 allora $\psi(x_1, \ldots, x_n) = 1$;
3. ...
   ⋮

Intuitivamente, potrei non finire mai o perché per ogni valore di $y$ esiste un $m_y$ che $\varphi(x_1, \ldots, x_n, y) = m_y \neq 0$, o perché per i primi $k$ numeri naturali $\varphi(x_1, \ldots, x_n, z) = n_z \neq 0$ e $\varphi(x_1, \ldots, x_n, k) \uparrow$. Infatti nel primo caso continuiamo a calcolare la $\varphi(x_1, \ldots, x_n, y)$ per valori crescenti di $y$ senza terminare mai, e nel secondo caso non ci arrestiamo mai nel calcolo di $\varphi(x_1, \ldots, x_n, k)$: da qui la parzialità di $\psi$. Se dovessi scrivere un programma, userei un comando di tipo `while`.

Attenzione, però! se qui l'intuizione ci aiuta, non dobbiamo affidarci esclusivamente a essa, ma procedere con il dovuto rigore matematico e la precisione che lo studio richiede — razzolo male, ma predico bene!

Vediamo adesso una semplicissima definizione $\mu$-ricorsiva, tramite la quale rappresentiamo l'archetipo delle funzioni parziali: la funzione ovunque indefinita, che è calcolabilissima!

**Esempio 1.8.2.** La seguente è una delle possibili derivazioni per la funzione ovunque indefinita $\psi_\uparrow(x)$:

$$\varphi = \lambda x, y.3$$

$$\psi_\uparrow = \lambda x.\, (\mu y.\varphi(x, y) = 0)$$

Per verificare che quanto scritto sopra è davvero una derivazione basta controllare che la funzione (ricorsiva primitiva) $\varphi$ è definita per tutti i suoi argomenti, ovvero che il calcolo di $\varphi(x, y)$ termina per ogni $x$; il che è banale. In questo caso è altrettanto facile vedere che per nessun $x$ esiste un $y$ per cui $\varphi(x, y) = 0$ e che quindi la funzione $\psi(x)$ è indefinita per ogni valore di $x$; attenzione però: nella maggior parte dei casi questo controllo è molto molto difficile, in un senso che sarà precisato formalmente tra poco.

Ricapitolando: si comincia con le ricorsive primitive, si applica l'operatore di minimizzazione $\mu$ e si ottiene una funzione $\mu$-ricorsiva, che può essere ora usata nella prossima definizione. Tutto ciò a patto che la condizione (*) valga, altrimenti si può uscire dalla classe, cioè, se $\varphi$ non termina per qualche valore $z$ minore del minimo $y$ su cui $\varphi$ vale 0, allora la funzione $\psi$ potrebbe non essere $\mu$-ricorsiva: terminazione e non terminazione sono *importantissime*!

Si noti anche che

$$f(x) = \begin{cases} \mu y[y < g(x), h(x, y) = 0] & \text{se esiste tale } y \\ 0 & \text{altrimenti} \end{cases}$$

è ricorsiva primitiva se $g$ e $h$ lo sono. La ragione è che $g$ impone un limite ai tentativi di ricercare il minimo $y$, e quindi o lo troviamo in meno di $g(x)$ applicazioni di $h$ o diamo come risultato 0 dopo al piú $g(x)$ applicazioni di $h$, che è un numero determinabile in tempo *finito* perché sia $g$ che $h$ sono totali in quanto ricorsive primitive. In altre parole anche la $f$ sarebbe totale, e quindi avremmo definito un formalismo che definisce solo funzioni totali e quindi inadatto a rappresentare *tutte* le funzioni calcolabili.

Infine, si suggerisce al lettore di verificare che la condizione (*) può essere rimossa, a condizione che la funzione $\varphi(x_1, \ldots, x_n, y)$ sia essa stessa una funzione ricorsiva primitiva.

Diamo ora alcune definizioni ausiliarie che ci saranno utili in seguito.

**Definizione 1.8.3.** Una relazione $I \subseteq \mathbb{N}^n$, $n \geq 1$ è *ricorsiva* (come sinonimo di totale) (o rispettivamente è *ricorsiva primitiva*, ha la proprietà $P$) se la sua funzione caratteristica $\chi_I$ è ricorsiva totale (è ricorsiva primitiva, ha la proprietà $P$).
Un caso particolare interessante si ha con gli *insiemi ricorsivi* $I \subseteq \mathbb{N}$, cioè quando $n = 1$.

In analogia a quanto fatto con le funzioni T-calcolabili, diciamo che una funzione è *$\mu$-calcolabile* se la sua definizione è $\mu$-ricorsiva.

Adesso abbiamo le funzioni T-calcolabili, quelle *WHILE*-calcolabili e quelle $\mu$-calcolabili e il bello è che formano *esattamente la stessa classe di funzioni calcolabili*, ciò che è stato accuratamente dimostrato. Abbiamo già annunciato che molti altri formalismi sono stati proposti e che tutti questi (quando siano sufficientemente potenti in un senso che renderemo preciso tra poco) definiscono la *stessa classe di funzioni*; in altre parole sono *Turing equivalenti*. Pertanto possiamo, o meglio vogliamo stipulare come vera la

| **Tesi di Church-Turing**: Le funzioni *(intuitivamente)* calcolabili sono tutte e sole le funzioni (parziali) T-calcolabili. |
|---|

In realtà questa è un'ipotesi, ma è talmente forte che la prendiamo come tesi. In termini informatici, questo significa che non importa quale linguaggio di programmazione usiamo, né su quale macchina facciam girare i nostri programmi, purché si abbia a disposizione memoria e tempo illimitati: ciò che possiamo calcolare *non* cambia — può forse cambiare *come* si lo si calcola.

Chiaramente è dimostrabile solo l'equivalenza tra i formalismi esistenti, ed è certamente molto difficile immaginare una dimostrazione di equivalenza tra tutti i *possibili* formalismi, inclusi quelli ancora da inventare.

La tesi di Church-Turing postula che la nozione di calcolabilità "intuitiva" è *robusta*. Inoltre, la tesi cade se si rilascia anche una sola delle ipotesi fatte sulla natura degli algoritmi.

Bene, di qui in avanti parleremo solo di *funzioni calcolabili*, senza qualificare ulteriormente il formalismo usato per definirle. Quante sono? E ce ne sono di non calcolabili? Se sí, ne vedremo una interessante?

## 1.9 Alcuni risultati classici

Introdurremo brevemente alcuni risultati basilari della teoria della calcolabilità che ne illustrano l'essenza, caratterizzando la classe delle funzioni, ovvero dei problemi calcolabili, mediante alcuni teoremi di "chiusura". Privilegeremo una presentazione orientata ai fondamenti dell'informatica, a volte purtroppo senza la profondità e l'accuratezza che sarebbero necessari nel presentare una teoria in cui precisione e attenzione ai dettagli giocano un ruolo essenziale. Prima di enunciare questi risultati, insistiamo a ricordare che, grazie alla tesi di Church, possiamo chiamare *calcolabili* indifferentemente le funzioni esprimibili nel formalismo delle macchine di Turing o le funzioni $\mu$-ricorsive o i programmi WHILE o ciò che volete voi, purché la loro definizione rispetti le cinque condizioni intuitive poste agli algoritmi che sono state espresse nel primo capitolo: le nostre ipotesi di lavoro.

Cominciamo con un semplice risultato sulla cardinalità [18] dell'insieme delle funzioni calcolabili, da cui segue che vi sono funzioni *non calcolabili*.

**Teorema 1.9.1.**

*i) Le funzioni calcolabili sono $\#(\mathbb{N})$; inoltre anche le funzioni calcolabili totali sono $\#(\mathbb{N})$*

*ii) esistono funzioni non calcolabili.*

*Dimostrazione.*

i) Costruisci $\#(\mathbb{N})$ MdT $M_i$ che svuotano il nastro dall'input, ci scrivono la stringa $|^i$ e si arrestano (sono tutte funzioni costanti). Che non siano piú di $\#(\mathbb{N})$ segue dal fatto che le MdT si possono enumerare, come fatto intuire a pagina 35.

ii) Con una costruzione analoga a quella di Cantor (la classe dei sottoinsiemi di $\mathbb{N}$ non è numerabile) si vede che $\{f : \mathbb{N} \to \mathbb{N}\}$ ha cardinalità $2^{\#(\mathbb{N})}$. □

Abbiamo già visto nel capitolo 1.6 che possiamo associare un indice alle macchine di Turing codificandole (veramente avevamo una funzione solamente iniettiva, ma sappiamo che Gödel ne ha definito una anche surgettiva); analogamente nel capitolo successivo abbiamo accennato a come enumerare le funzioni ricorsive primitive e non è difficile immaginare una sua estensione

[18] Dato un insieme $A$, indicheremo con $\#(A)$ la sua cardinalità, ovvero il numero dei suoi elementi.

alle funzioni $\mu$-ricorsive. Oltre alla superficialità con cui sono stati presentati, i due modi hanno in comune il fatto che si basano *solamente* sui simboli usati nel definire gli algoritmi, il che è bene.

Infatti, sotto ipotesi molto ragionevoli, per i nostri scopi non c'è sostanziale differenza tra una enumerazione e un'altra, purché sia *effettiva*. Quindi siamo liberi di scegliere quella che piú ci aggrada. Per saperne di più e avere una definizione formale delle enumerazioni effettive, si veda, ad esempio il [Rogers]. Basti qui dire che una buona enumerazione deve essere una funzione *calcolabile totale biunivoca* che dipende *solo* dalla sintassi con cui scriviamo gli algoritmi e *non* dal significato che attribuiamo loro. Di nuovo, la già ricordata enumerazione del capitolo 1.6 andrebbe bene se fosse surgettiva; invece una enumerazione che pretendesse, ad esempio, di elencare prima tutte le funzioni costanti e poi le altre non sarebbe effettiva. L'osservazione appena fatta ci consente anche di fissare una volta per tutte un elenco di MdT (programmi *WHILE*, funzioni $\mu$-ricorsive, ...) e di indicare con $M_i$ la MdT (il programma, la funzione $\mu$-ricorsiva, ...) che vi appare in posizione $i$-ma, o meglio *l'algoritmo* $i$-mo. Ancor meglio, si può usare la seguente notazione, che finalmente evidenzia la differenza tra *funzione* e *algoritmo* che la calcola.

**NOTAZIONE** Data un'enumerazione effettiva, indicheremo con $\varphi$ la funzione (parziale) che la macchina, o meglio l'algoritmo, $M_i$ calcola e chiameremo $i$ *indice* (non della funzione, bensí della macchina! quindi può darsi benissimo che per $i \neq j$ sia $\varphi_i = \varphi_j$, mentre sicuramente $M_i \neq M_j$).

Entriamo adesso nel vivo della presentazione dei teoremi piú importanti di questa prima parte del corso, ribadendo ancora una volta il seguente fatto.

**Repetita:** I risultati che riportiamo nel seguito sono tutti *invarianti* rispetto all'enumerazione scelta.

Il primo teorema, che spesso è chiamato *padding lemma*, ci dice che ci sono infinite (numerabili) MdT, ovvero infiniti numerabili algoritmi che calcolano la stessa funzione, e che *alcuni* di essi si possono costruire "facilmente" da un algoritmo dato (ossia esprimibile con una funzione primitiva ricorsiva o un programma *FOR*).

**Teorema 1.9.2.** *Ogni funzione calcolabile $\varphi_x$ ha $\#(\mathbb{N})$ indici. Inoltre $\forall x$ si può costruire, mediante una funzione ricorsiva primitiva, un insieme infinito $A_x$ di indici tale che*

$$\forall y \in A_x.\ \varphi_y = \varphi_x$$

*cioè $\varphi_y(n) = m$ sse $\varphi_x(n) = m$ (e ovviamente $\varphi_y(n) \uparrow$ sse $\varphi_x(n) \uparrow$).*

*Dimostrazione.* Per ogni macchina $M_x$, se $Q = \{q_0, \ldots, q_k\}$, ottieni la prima macchina $M_{x_1}$ con $x_1 \in A_x$ aggiungendo lo stato $q_{k+1} \notin Q$ e la quintupla $(q_{k+1}, \#, q_{k+1}, \#, -)$; ottieni la seconda $M_{x_2}$ aggiungendo lo stato $q_{k+2}$ e la quintupla $(q_{k+2}, \#, q_{k+2}, \#, -)$, .... □

Il prossimo teorema dice che tra tutti gli algoritmi che calcolano una data funzione ce n'è uno privilegiato, nel senso che ha una forma speciale. Di conseguenza, ogni funzione ha una rappresentazione privilegiata.

**Teorema 1.9.3** (forma normale)**.** *Esistono un predicato $T(i, x, y)$ e una funzione $U(y)$ calcolabili totali tali che $\forall i, x.\ \varphi_i(x) = U(\mu y.\, T(i, x, y))$.*
*Inoltre $T$ e $U$ sono primitive ricorsive.*

*Dimostrazione.* Definisci $T(i, x, y)$, detto comunemente *predicato di Kleene*, vero se e solamente se $y$ è la codifica di una computazione terminante di $M_i$ con dato iniziale $x$. Per calcolare $T$, dato $i$ recupera $M_i$ dalla lista e comincia a scandire i valori $y$. Decodifica ognuno di essi, uno alla volta, e, avendo come ingresso $x$ controlla se il risultato è una computazione terminante della forma $M_i(x) = c_0, c_1, \ldots, c_n$. Questa sequenza di passi termina sempre e quindi $T$ è totale. Se $y$ è la codifica di una computazione terminante di $M_i(x)$, allora $c_n = (h, \triangleright z\underline{\#})$ e definisci $U$ in modo che $U(y) = z$ e i passi necessari a questo calcolo sono finiti e terminano tutti: quindi $U$ è totale. L'intero procedimento seguito è effettivo, e quindi $T$ e $U$ sono calcolabili per la tesi di Church-Turing.
Inoltre $U$ e $T$ sono ricorsivi primitivi perchè le codifiche che usiamo lo sono e perché lo sono i controlli effettuati. □

Si noti che la versione del teorema di forma normale sulle MdT deterministiche è tale per cui se esiste un $y$ tale che $T(i, x, y)$ risulta vero, allora tale $y$ è anche unico. Invece, se le MdT considerate fossero non-deterministiche (vedi Def. 3.2.1), ci potrebbero essere più computazioni che terminano nello stato $h$, per cui l'operatore di minimizzazione darebbe come risultato il minimo intero che codifica una di esse (si veda la discussione sulle regole di valutazione fatta nel capitolo 1.6).

Una immediata conseguenza del teorema di forma normale è che ogni funzione calcolata da una MdT ammette una definizione $\mu$-ricorsiva. In altre parole, vale il seguente corollario.

**Corollario 1.9.4.** *Le funzioni T-calcolabili sono $\mu$-ricorsive.*

Se c'è tempo, nelle esercitazioni costruiremo le macchine per gli schemi di definizione I-III (facile) e per gli schemi IV, V e VI (meno facile). Cosa noiosa assai. Ciò costituisce la dimostrazione costruttiva del seguente lemma.

**Lemma 1.9.5.** *Le funzioni $\mu$-calcolabili sono T-calcolabili.*

Ora è facile concludere l'equivalenza tra MdT e funzioni $\mu$-ricorsive.

**Teorema 1.9.6.**
*Una funzione è T-calcolabile se e solamente se è $\mu$-calcolabile.*

Il teorema di forma normale e quello d'equivalenza tra MdT e funzioni $\mu$-ricorsive ha il seguente corollario interessante dal punto di vista informatico. La sua rilevanza nel nostro campo è legata al fatto che le funzioni primitive ricorsive si possono rappresentare con un programma nel linguaggio *FOR* e quelle $\mu$-ricorsive con uno nel linguaggio *WHILE* (v. capitolo precedente); pertanto, si deduce che *ogni programma* può essere scritto (in forma normale) usando un comando di tipo `while` e due di tipo `for` (ciò è particolarmente rilevante quando il programma in questione sia l'interprete di un linguaggio, come vedremo piú avanti).

**Corollario 1.9.7.** *Ogni funzione calcolabile parziale può essere ottenuta da due funzioni primitive ricorsive e una sola applicazione dell'operatore $\mu$.*

Adesso arriva un teorema molto importante. Dice che un formalismo universale, cioè uno che esprima *tutte* le funzioni calcolabili, è cosí potente da riuscire ad esprimere *l'interprete dei propri programmi*. Vedremo piú avanti che questa capacità può essere usata "alla rovescia", cioè per mostrare che un formalismo è universale. Vediamo il semplice caso con funzioni a una variabile e la sua dimostrazione; l'estensione al caso generale per funzioni a $n$ variabili è immediata.

**Teorema 1.9.8** (enumerazione)**.** *Esiste una funzione calcolabile parziale $\varphi_z(i,x)$ tale che $\varphi_z(i,x) = \varphi_i(x)$.*
*(Si noti l'ordine dei quantificatori: $\exists z$ tale che $\forall i, x$ si ha $\varphi_z(i,x) = \varphi_i(x)$.)*

*Dimostrazione.* Poiché la funzione $U(\mu y.\, T(i,x,y))$ usata nel teorema di forma normale è definita per composizione e $\mu$-ricorsione a partire da funzioni (meglio da una funzione e un predicato) primitive ricorsive, essa stessa è una funzione calcolabile in due argomenti $i$ e $x$. Avrà quindi un indice che chiamiamo $z$, cioè sia $\varphi_z(i,x) = U(\mu y.\, T(i,x,y))$. Applichiamo allora il teorema di forma normale, da cui $U(\mu y.\, T(i,x,y)) = \varphi_i(x)$. Per ottenere la tesi basta la transitività dell'uguaglianza.
Piú intuitivamente e informalmente: $M_z$ recupera la descrizione di $M_i$ e la applica a $x$. □

Il teorema di enumerazione garantisce che esiste la MdT Universale, $M_z$, che ha in ingresso la descrizione di una MdT $M_i$, il dato $x$ e "si comporta come $M_i$". [19] E' esattamente quanto avviene ogni giorno con i nostri programmi: li diamo in pasto a una macchina realizzata in parte H/W, in parte S/W (si tratta di una macchina reale, e quindi solo "quasi-universale", perchè, tra l'altro, ha memoria limitata). La macchina in questione esegue i nostri programmi, ovvero si comporta esattamente come dettato dai suoi dati, i programmi in ingresso: finché l'istruzione corrente non è STOP prende i suoi argomenti; esegue l'istruzione; ne memorizza il risultato; aggiorna il puntatore all'istruzione corrente (cf. la dimostrazione del teorema di enumerazione, l'enunciato del teorema di forma normale e la breve discussione dopo il teorema 1.9.6).

Intuitivamente, il teorema di enumerazione ci "libera" dalla necessità di avere un esecutore umano delle Mdt, cosí come previsto da Alan Turing: *esiste una macchina che esegue gli algoritmi.*

Ma com'è fatta una MdT universale? Ne vediamo una descrizione, per cosí dire, a tratteggio, rimandando il lettore alla letteratura per una definizione accurata (p.e. al libro di Aiello, Albano, Attardi, Montanari).

Usiamo una variante delle MdT in cui sono previsti tre nastri, ciascuno semi-infinito a destra. Questo tipo di macchine verranno usate nella terza parte del corso e saranno formalmente introdotte nella definizione 3.2.1. Ovviamente il potere espressivo delle MdT cosí fatte non cambia (si veda anche il Teorema 3.2.6). Avere tre nastri ci libera dal dover partizionare il nastro unico in tre pezzi, ben separati tra loro, alcuni dei quali devono essere spostati in blocco quando la testina abbia bisogno di "spazio" o quando si cancelli l'ultimo simbolo a destra di un pezzo.

Supponiamo di avere i seguenti insiemi ausiliari (N.B. non sono gli stati e i simboli della MdT universale):

$$Q_* = \{q_0, q_1, \dots\} \text{ con } h \notin Q_* \text{ e } \Sigma_* = \{\sigma_0, \sigma_1, \dots\} \text{ con } L, R, - \notin \Sigma_*$$

in questo modo, ogni MdT $M_k$ ha l'insieme degli stati $Q_k$ e quello dei simboli $\Sigma_k$ inclusi rispettivamente in $Q_*$ e $\Sigma_*$. Andiamo adesso a rappresentare gli elementi di $Q_*$ e $\Sigma_*$ (ovvero gli stati e i simboli di tutte le $M_k$) come stringhe del monoide $\{|\}^*$.

$$\kappa : Q_* \cup \{h\} \cup \Sigma_* \cup \{L, R, -\} \to \{|\}^*$$
$$q_i \mapsto |^{i+2} \quad h \mapsto |$$
$$\sigma_i \mapsto |^{i+4} \quad L \mapsto | \quad R \mapsto || \quad - \mapsto |||$$

La funzione di codifica $\kappa$ non è biunivoca, ma lo è quando sappiamo se la stringa che dobbiamo decodificare è uno stato o un simbolo o una

[19] A titolo di curiosità: esistono MdTU "compatte" con $(24, 2)$, $(10, 3)$, $(7, 4)$, $(5, 5)$, $(4, 6)$, $(3, 10)$ e $(2, 18)$, dove $(m, n)$ indica la coppia (numero di stati, numero di simboli).

direzione. Pertanto separereremo la codifica degli stati da quella dei simboli e delle direzioni con un'opportuna marca $c$ non appartenente all'alfabeto della macchina da codificare.

Fissiamo adesso una MdT $M = (Q, \Sigma, \delta, s)$ ($s \in Q$ è lo stato iniziale). La prima cosa da fare per codificarla è ordinare totalmente l'insieme degli stati $Q = \{q_{i_1}, \ldots, q_{i_k}\}$ e l'insieme dei simboli $\Sigma = \{\sigma_{j_1}, \ldots, \sigma_{j_\ell}\}$ in modo che $p \leq p' \Rightarrow i_p \leq i_{p'}$ e $j_p \leq j_{p'}$.

Consideriamo ora l'alfabeto $\{|, c, d, \#, \triangleright\}$, sotto l'ipotesi che $\{|, c, d\} \cap (\Sigma_* \cup Q_*) = \emptyset$, e codifichiamo ogni quintupla di $M$ così:

$$S_{p,q} = c\ \kappa(q_{i_p})\ c\ \kappa(\sigma_{j_q})\ c\ \kappa(q)\ c\ \kappa(\sigma)\ c\ \kappa(D)\ c \quad \text{se } \delta(q_{i_p}, \sigma_{j_q}) = (q, \sigma, D)$$

Infine, completiamo l'operazione codificando *anche* i casi in cui $\delta$ non è definita, per maggior chiarezza nel passo successivo di codifica. Quindi poniamo

$$S_{p,q} = c\ \kappa(q_{i_p})\ c\ \kappa(\sigma_{j_q})\ c\ d\ c\ d\ c\ d\ c \quad \text{se } \delta(q_{i_p}, \sigma_{j_q}) \text{ non è definita}$$

**Esempio 1.9.9.** Consideriamo la MdT $\widehat{M} = (\{q_2\}, \{\sigma_1, \sigma_3, \sigma_5\}, \delta, q_2)$, dove

$$\begin{array}{lcll} \delta(q_2, \sigma_1) & = & (h, \sigma_5, -) & \mapsto S_{1,1} \\ \delta(q_2, \sigma_3) & = & (q_2, \sigma_1, R) & \mapsto S_{1,2} \end{array}$$

Allora $k = 1$ ($i_1 = 2$) e $\ell = 3$ ($j_1 = 1, j_2 = 3, j_3 = 5$) e $S_{1,1}$, $S_{1,2}$ e $S_{1,3}$ (quest'ultima per il caso $\delta(q_2, \sigma_5)$ indefinito) sono definiti cosí

$$\begin{array}{l} S_{1,1} = c\ |^4\ c\ |^5\ c\ |\ c\ |^9\ c\ |^3\ c \\ S_{1,2} = c\ |^4\ c\ |^7\ c\ |^4\ c\ |^5\ c\ |^2\ c \\ S_{1,3} = c\ |^4\ c\ |^9\ c\ d\ c\ d\ c\ d\ c \end{array}$$

Ritorniamo al problema di definire una funzione di codifica $\rho$ che mandi una generica MdT $M$ in una stringa fatta da soli caratteri $|, c$ e $d$, che verrà racchiusa tra $c$. Ciò che manca è di sapere qual è lo stato iniziale di $M$, che codificheremo e prefiggeremo alla codifica della funzione di transizione $\delta$, nel modo seguente:

$$\rho(M) = c\ \kappa(s)\ c\ S_{1,1}S_{1,2} \ldots S_{1,\ell}S_{2,1} \ldots S_{2,\ell} \ldots S_{k,1}S_{k,2} \ldots S_{k,\ell}\ c$$

È facile vedere che si può tornare indietro in modo univoco da $\{|, c, d\}^*$ o a una MdT o a qualcosa che non lo è, se la stringa non è immagine di una qualche MdT (ancora una volta, ciò non ci preoccupa, anche se non è molto bello, perché si potrebbe far meglio, al prezzo di una funzione di codifica assai piú complicata). Per tornare indietro, nella stringa $c\ w_0\ c\ \ c\ w_1 w_2 \ldots$ decodifica $w_0$ per trovare $s$, poi da $w_1$ ottieni la prima quintupla (se $w_1$ è in forma corretta) o nulla se $w_1 = |^n\ c\ |^m\ c\ d\ c\ d\ c\ d\ c$, eccetera.

**Esempio 1.9.10.** Riprendiamo l'esempio di sopra e vediamo la codifica della intera $\hat{M}$

$c\,|^4\,c\;\;c\,|^4\,c\,|^5\,c\,|\,c\,|^9\,c\,|^3\,c\;\;c\,|^4\,c\,|^7\,c\,|^4\,c\,|^5\,c\,|^2\,c\,c\,|^4\,c\,|^9\,c\,d\,c\,d\,c\,d\,c\,c$

Si noti che la stringa inizia e finisce con $c$ e tra ogni gruppo (stato iniziale o quintupla) ci sono sempre due $c$, quindi si riesce a determinare sempre quale interpretazione dare a una stringa di $|$.

Naturalmente non abbiamo ancora finito, perché abbiamo detto che la MdT Universale, che chiamiamo $U$, deve ricevere la descrizione di una MdT $M$ e del suo dato iniziale $w = \sigma'_0 \dots \sigma'_n$. Ecco l'ultimo passo della codifica:

$$\rho(M)\tau(w)$$

dove

$$\tau(\sigma'_0 \dots \sigma'_n) = c\;\kappa(\sigma'_0)\;c\;\kappa(\sigma'_1)\;c\;\dots\kappa(\sigma'_n)\;c.$$

Si noti che tre $c$ separano la codifica di $M$ da quella di $w$.

Quello che vogliamo dalla MdT universale $U = (Q_U, \Sigma_U, \delta_U, s_U)$ è che per ogni $M = (Q, \Sigma, \delta, s)$ e per ogni $w \in \Sigma^*$ succeda che

i) se $(s, \trianglerighteq w) \to^*_M (h, u\underline{a}v)$ allora $(s_U, \trianglerighteq \rho(M)\tau(w)) \to^*_U (h, \tau(uav)\underline{\#})$

ii) se $(s_U, \trianglerighteq \rho(M)\tau(w)) \to^*_U (h, u'\underline{a'}v')$ allora $a' = \#$, $v' = \epsilon$ e $u' = \tau(uav)$, per qualche $u$, $a$ e $v$ tali che $(s, \trianglerighteq w) \;\to^*_M\; (h, u\underline{a}v)$

Cioè

i) se $M$ si ferma su $w$ con risultato $uav$, allora $U$ si ferma su $\rho(M)\tau(w)$ e (la porzione a sinistra del cursore su) il nastro è la codifica di $uav$.

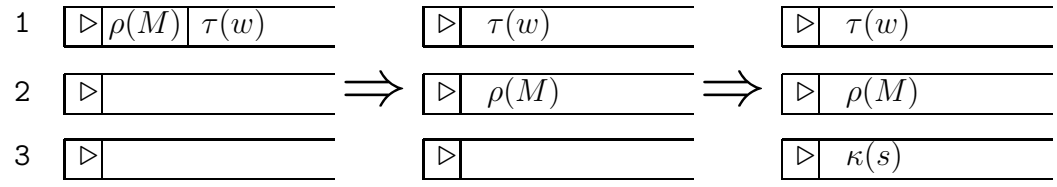
ii) viceversa.

Si noti che il comportamento della macchina universale $U$ non è rilevante se il dato di ingresso non è nella forma $\rho(M)\tau(w)$.

La nostra MdT universale $U$ ha 3 nastri; come detto sopra, questa non è una limitazione reale, e serve a semplificare la descrizione che diamo di seguito. Se si desiderasse avere una MdT con un solo nastro, basterebbe giustapporre i tre nastri, separandoli con un'opportuna marca e poi spostare a destra o a sinistra porzioni di nastro e/o nastri interi qualora ve ne fosse la necessità (vedremo a spanne come fare nella dimostrazione del teorema 3.2.6). A regime, nel primo nastro c'è la codifica $\tau(w)$ del nastro $w$ della macchina $M$ argomento della $U$, nel secondo c'è la codifica di $\rho(M)$ della macchina in questione, nel terzo c'è la codifica dello stato in cui si trova $M$. In parole piú informatiche, il secondo nastro rappresenta il programma da eseguire, il

primo rappresenta i dati da elaborare, il terzo è una componente del contatore istruzioni (l'altra sarà il simbolo corrente).

Prima di "eseguire" $M$ su $w$, la MdT universale esegue alcuni passi di inizializzazione: porta sul secondo nastro la codifica di $M$, sul terzo la codifica del suo stato iniziale $s$ e lascia sul primo la codifica di $w$ (vedi disegno).




Dopo le azioni di inizializzazione descritte nel disegno, possiamo cominciare a tratteggiare il comportamento di $U$, chiamando testina $i$ il cursore sull'$i$-esimo nastro:

i) Sposta la testina 2 sul primo carattere di $S_{i,1}S_{i,2}\dots S_{i,\ell}$ se il nastro 3 contiene la codifica di $q_{h_i}$. (Questo si può fare per confronti e spostamenti "simultanei".)

ii) Sposta la testina 2 sulla "quintupla" $S_{i,j}$ se la testina 1 è all'inizio di $\kappa(\sigma_{k_j})$. (Come sopra, per confronti e spostamenti "simultanei")

iii) Se $S_{ij} = c\ |^p\ c\ |^q\ c\ |^{p'}\ c\ |^{q'}\ c\ |^r\ c$

- scrivi sul terzo nastro $|^{p'}$ al posto di $\kappa(q_{h_i})$;
- scrivi $|^{q'}$ sul primo nastro al posto di $\kappa(\sigma_{k_j})$, eventualmente spostando a destra o a sinistra la porzione di nastro a destra di $\kappa(\sigma_{k_j})$, a seconda che la lunghezza di $\kappa(\sigma_{k_j})$ sia minore o maggiore di quella di $|^{q'}$;
- sposta la testina 1 al precedente o al successivo gruppo $c|^v c$, oppure lasciala dov'è, a seconda del valore di $r$.

Se $S_{ij} = c\ |^p\ c\ |^q\ c\ d\ c\ d\ c\ d\ c$ fermati (condizione d'errore).
Se $S_{ij} = c\ |^p\ c\ |^q\ c\ |\ c\ |^{q'}\ c\ |^r\ c$ fermati (con successo) dopo aver portato il cursore sul primo # che si incontra a destra.

Se confrontiamo il comportamento della macchina universale $U$ con quello di un "normale" interprete di un linguaggio di programmazione, si nota che

- i passi (i) e (ii) corrispondono al reperimento dell'istruzione corrente: attraverso il contenuto del terzo nastro e mediante il simbolo corrente si realizza infatti il "contatore istruzioni";
- il passo (iii) incorpora la decodifica dell'istruzione corrente e, se questa non è un errore o l'istruzione di fine esecuzione, l'esecuzione vera e propria dell'istruzione, con relativa memorizzazione del risultato.

Ritorniamo adesso ai risultati classici, presentando il teorema del parametro, detto anche teorema *s-m-n*; vediamo prima il caso in cui $m = n = 1$.

**Teorema 1.9.11** (s-1-1)**.** *Esiste una funzione calcolabile totale (iniettiva)* $s_1^1$ *con due argomenti, tale che* $\forall i, x$

$$\varphi_{s_1^1(i,x)} = \lambda y.\ \varphi_i(x, y)$$

Intuitivamente, la macchina $M_{s(i,x)}$, o piú liberamente l'algoritmo o il programma $P_{s(i,x)}$ (omettiamo d'ora in poi l'indice e l'apice 1, per leggibilità) opera su $y$ soltanto, mentre $P_i$ opera su $x$ e su $y$. Quindi $x$ è un *parametro* di $P_i$. Ad esempio, sia $\varphi_i(x, y)$ la funzione $x \times f(y)$ (con $f$ qualunque); allora, a partire da $i$ e da 2, mediante la $s$ trovo in *modo effettivo* l'indice del programma che calcola la funzione $2 \times f(y)$, cioè determino $\varphi_{s(i,2)}$.

Il teorema *s-m-n* è importante in informatica perché è la base per la tecnica di "valutazione parziale" secondo la quale si specializza via via un programma generale per ottenerne versioni piú efficienti in casi particolari (compilatori/interpreti per architetture parametriche, ecc.). Questo teorema è inoltre utilissimo nella teoria della calcolabilità sia perché ci offre uno strumento potentissimo da usare nelle dimostrazioni, sia per le ragioni espresse dal teorema di espressività, riportato piú avanti. Vediamo la dimostrazione della sua versione *s-1-1*, cioè con $m, n = 1$, e poi l'enunciato generale.

*Dimostrazione intuitiva del teorema s-1-1*
Per calcolare $\varphi_{s_1^1(i,x)}(y)$ prendi $M_i$ decodificando $i$ e predisponi lo stato iniziale, cioè $M_i(x, y)$, dove $x$ è fissato in anticipo. (Se vi fosse piú familiare, prendete il programma $P_i$ che avrà un'istruzione per leggere il valore $k$ del parametro $x$: allora rimuovetela e inserite al suo posto l'assegnamento $x := k$.) Quella delineata (in entrambi i casi) è una procedura effettiva, cioè algoritmica, che termina sempre, quindi per la tesi di Church-Turing esiste una funzione calcolabile totale $s = s_1^1$. Se tale funzione non fosse iniettiva, allora costruisci $s'$ con $\varphi_{s(i,x)} = \varphi_{s'(i,x)}$ in modo tale che $s'(i, x)$ generi indici (che esistono perché gli indici delle MdT che calcolano $s(i, x)$ sono $\#(\mathbb{N})$) in modo strettamente crescente, cioè tali che $s'(i_0, x_0) > s'(i_1, x_1)$ se la codifica della coppia $(i_0, x_0)$ è maggiore di quella di $(i_1, x_1)$. A questo punto basta notare che una funzione totale strettamente crescente è iniettiva. □

Ritorniamo all'esempio precedente, in cui veniva considerata la funzione $y \times f(z)$ e vediamo, in modo molto approssimativo e intuitivo e senza alcuna pretesa di completezza, come può essere fatta la funzione che trasforma un programma che calcola $x \times f(y)$ in uno (piú efficiente) che calcola $2 \times f(y)$, ossia come può esser fatta la funzione $s$.

Innanzitutto, (ri-)definiamo il prodotto tra due naturali $x$ e $w$, mediante il seguente programma nel linguaggio *WHILE*:

```
prodotto := 0;
while x > 0 do
   prodotto := prodotto + w;                          PROGRAMMA 1
   x := x-1
```

Inoltre, supponiamo di avere a disposizione un "valutatore parziale" (e simbolico) in grado usare l'interprete del linguaggio, ovvero la sua semantica operazionale. Questo deve essere in grado di eseguire i comandi nel giusto ordine; di estrarre, quando possibile, il valore corrente delle variabili (cioè di leggere il contenuto delle caselle di memoria); e infine di eseguire alcune semplici operazioni "primitive," sia aritmetiche che di confronto. Supponiamo anche di avere a disposizione un comando che scrive su un nastro di uscita una stringa di caratteri, che poi interpreteremo come un programma.

Naturalmente, il calcolo di $\varphi_i(x, y)$ si riduce a quello del prodotto di 2 e di $f(y)$. Allora procediamo a valutare parzialmente il programma scritto sopra, scrivendo via via sul nastro di uscita i comandi che andremo a eseguire.

Il primo passo è guidato dalla semantica operazionale del ";" e consiste nell'inizializzazare `prodotto`, e scrivere tale comando, poiché `prodotto` è di uscita (se non lo fosse potremmo eliminare il comando `1` e semplificare il `2`):

1. `prodotto := 0;`

A questo punto bisogna valutare il comando `while`, la cui semantica ci richiede di valutare un comando `if-then-else`, ma la condizione è vera, poiché la variabile `x` contiene il valore 2. Quindi in due passi, senza toccare il valore delle variabili, passiamo a valutare

```
(prodotto := prodotto + w;
x := x-1);
(while x > 0 do
   prodotto := prodotto + w;
   x := x-1)
```

La valutazione del primo comando richiede di incrementare il valore della variabile `prodotto` e di scrivere in uscita il comando eseguito, sostituendo a `w` il suo valore $f(y)$:

2. `prodotto := prodotto +` $f(y)$`;`

Poi va decrementato il valore della variabile `x`, ma non trascriviamo il comando in uscita perché questa è proprio la variabile che consideriamo come parametro. Si passa allora a valutare nuovamente il `while`, poi l'`if-then-else` la cui condizione `x > 0` è ancora vera perché `x` contiene 1, e analogamente a quanto fatto prima si scrive sul nastro di uscita il comando

3. `prodotto := prodotto +` $f(y)$`;`

e ad `x` viene assegnato 0. La semantica del comando `while` ci dice che dopo un passo non dobbiamo far piú nulla, perché 0 non è maggiore di 0.

A questo punto andiamo a leggere nel nastro di uscita (da cui abbiamo rimosso l'ultimo `;`) il programma che calcola proprio la funzione desiderata:

```
prodotto := 0;
prodotto := prodotto + f(y);                                  PROGRAMMA 2
prodotto := prodotto + f(y)
```

(Si poteva certamente far meglio: un semplicissimo ottimizzatore di codice di quelli presenti in ogni compilatore, avrebbe subito semplificato PROGRAMMA 2 usando il fatto che 0 è l'identità della somma, ottenendo:

`prodotto :=` $f(y) + f(y)$

o ancor meglio

```
prodotto := f(y);
prodotto := prodotto + prodotto
```

in cui la valutazione della funzione $f$ vien fatta una sola volta (cf. la discussione sulle regole di valutazione fatta a pagina 33).

In ogni caso, abbiamo ottenuto un programma eseguendo un numero *finito* di passi che hanno effetto *limitato*, in modo deterministico: abbiamo cioè definito un algoritmo che rappresenta proprio la nostra funzione $s$, che quindi è calcolabile e totale; detto in altri termini, abbiamo definito l'algoritmo che ci fa passare dal PROGRAMMA 1 al PROGRAMMA 2. Piú in generale la funzione $s(i, x)$ che ci permette di individuare nella lista dei programmi proprio uno di quelli che calcolano la funzione $\varphi_i(x, y)$ cercata avrà una struttura molto simile all'algoritmo appena delineato.

Nell'esempio appena concluso abbiamo usato un comando di tipo `while`, sebbene ciò non fosse strettamente necessario, in quanto è facile scrivere un comando di tipo `for` che realizzi il prodotto di due numeri per somme successive. In questo caso l'algoritmo che realizza la funzione $s(i, x)$ abbozzato sopra funziona bene, ma in generale ci potrebbero essere comandi `while` che in alcuni casi terminano e in altri no. Il valutatore parziale dovrà quindi essere assai piú astuto di quanto non abbiamo appena fatto vedere, per approssimare correttamente e per quanto possibile comportamenti non finiti; per maggiori dettagli e commenti su questo punto si veda la ricca letteratura sull'argomento [Jones].

Ecco adesso l'enunciato generale del teorema del parametro, in cui per maggior chiarezza scriviamo $\varphi_i^{(n)}$ per indicare che la funzione calcolabile $\varphi_i$ ha $n$ argomenti.

**Teorema 1.9.12** (s-m-n, o del parametro)**.**
$\forall m, n \geq 0$ *esiste una funzione calcolabile totale (iniettiva)* $s_n^m$ *con* $m+1$ *argomenti tale che* $\forall i, x_1, \ldots, x_m$

$$\varphi^{(n)}_{s_n^m(i,x_1,\ldots,x_m)} = \lambda y_1, \ldots, y_n.\ \varphi_i^{(m+n)}(x_1, \ldots, x_m, y_1, \ldots, y_n)$$

Si noti come il teorema del parametro e quello di enumerazione siano in un certo senso l'inverso l'uno dell'altro. Infatti l'uno "abbassa" un argomento nella posizione di indice, mentre l'altro "innalza" un indice nella posizione di argomento.

L'importanza dei teoremi del parametro e di enumerazione può essere compresa ancora meglio considerando il seguente teorema che non dimostreremo.

**Teorema 1.9.13** (espressività)**.** *Un formalismo è* Turing-equivalente *(calcola* tutte e sole *le funzioni T-calcolabili, è* universale*) se e solamente se*

- *ha un algoritmo* universale *(cioè vale il teorema di enumerazione)*
- *vale il teorema del parametro.*

Grazie al teorema *s-m-n* si dimostra un teorema molto elegante, che ha un ruolo fondamentale, sia in informatica che nella pura teoria della calcolabilità.

**Teorema 1.9.14** (ricorsione, Kleene II)**.** $\forall f$ *funzione calcolabile totale* $\exists n$ *tale che* $\varphi_n = \varphi_{f(n)}$*.*
*(Un tale indice viene detto* punto fisso *di* $f$*.)*

Prima della dimostrazione, diamo un un po' di intuizione: la funzione $f$ "trasforma" programmi in programmi, come fanno i compilatori. Infatti $f$ trasforma indici: dato $n$, ovvero il programma $P_n$ lo trasforma in $P_{f(n)}$. Quando consideriamo il punto fisso, la trasformazione operata da $f$ *non cambia* la funzione calcolata, ovvero trasforma un programma $P_n$ nel programma $P_{f(n)}$ con la *stessa* semantica. Si noti che l'accezione "punto fisso" usata qui è diversa da quella solita in cui il punto fisso $x$ di una funzione $g$ è tale che $g(x) = x$: qui il punto fisso riguarda la *funzione*, e non l'indice delle macchine che la calcolano, e che viene trasformato dalla funzione calcolabile totale $f$: si tratta di una trasformazione che conserva la semantica.

Questo teorema fornisce in un certo senso la "base" della semantica denotazionale; garantisce la realizzabilità di macchine che eseguono programmi ricorsivi o delle funzioni di crittografia o di molte altre diavolerie infernatiche. Nella dimostrazione del teorema si fa uso del fatto che le funzioni calcolabili sono chiuse rispetto alle trasformazioni di indici introdotte dal teorema del parametro.

*Dimostrazione.* Definiamo la seguente funzione calcolabile "diagonale"

$$\psi(u,z) = \varphi_{d(u)}(z) = \begin{cases} \varphi_{\varphi_u(u)}(z) & \text{se } \varphi_u(u)\downarrow \\ \text{indefinita} & \text{altrimenti} \end{cases} \qquad (1)$$

Per il teorema *s-m-n*, $d(u)$ è totale e iniettiva (e *non* dipende da $f$ ). [20] Data $f$, $f \circ d$ è calcolabile e sia $v$ proprio un indice tale che

$$\varphi_v(x) = f(d(x)) \qquad (2)$$

Tale funzione è totale (perché sia $d$ che $f$ lo sono), e quindi $\varphi_v(v)\downarrow$.
Pertanto, in accordo con la definizione (1) abbiamo $\varphi_{d(v)} = \varphi_{\varphi_v(v)}$. Calcoliamo adesso $d(v)$ e supponiamo che il risultato sia $n$, cioè poniamo

$$n = d(v) \qquad (3)$$

Dimostriamo che $n$ è un punto fisso di $f$. Infatti

$$\varphi_n \overset{(3)}{=} \varphi_{d(v)} \overset{(1)}{=} \varphi_{\varphi_v(v)} \overset{(2)}{=} \varphi_{f(d(v))} \overset{(3)}{=} \varphi_{f(n)}$$

Si noti come nell'eguaglianza piú a sinistra si sfrutti l'iniettività della funzione $d$, garantitaci dal teorema del parametro. □

Ci sono due fatti interessanti che sono correlati con il (secondo) teorema di ricorsione.

**Proprietà 1.9.15.** *Nelle ipotesi del teorema di ricorsione,*

- *il punto fisso è calcolabile mediante una funzione totale (iniettiva) $g$ a partire da(ll'indice di) $f$;*
- *ci sono $\#(\mathbb{N})$ punti fissi di $f$.*

*Dimostrazione.* Diamo solo la traccia, senza considerare l'iniettività di $g$. Per dimostrare il primo punto si prenda $h(x)$ calcolabile totale tale che $\forall n$ $\varphi_{h(x)}(n) = \varphi_x(d(n))$. Allora $g(x) = d(h(x))$.
Il secondo punto segue dal Teorema 1.9.2. □

[20] Il teorema *s-m-n* in questo caso (e in molti altri) si applica con un po' di astuzia. Si scelga uno degli indici delle macchine che calcolano $\psi$, e lo si chiami $i$, cioè sia $\psi(u,z) = \varphi_i(u,z)$. A questo punto per il teorema del parametro si ha che $\varphi_i(u,z) = \varphi_{s(i,u)}(z)$, ma $s(i,u)$ dipende *solo* da $u$, e quindi ponendo $d(u) = \lambda u.\ s(i,u)$ si ottiene $\varphi_i(u,z) = \varphi_{d(u)}(z)$.

C'è un altro modo per dimostrare il teorema di ricorsione, o meglio per specificare come dev'essere implementata la ricorsione nei linguaggi di programmazione. Supponete di avere una procedura ricorsiva $P$ il cui corpo sia $C$, all'interno del quale ovviamente appare la chiamata a $P$ stessa. La tecnica usata per definire la semantica operazionale consiste nel memorizzare in un componente, di solito chiamato *ambiente* e rappresentato da una funzione $\rho$, l'associazione tra $P$ e $C$, cioè $\rho(P) = C$. Al momento della chiamata si cerca nell'ambiente il significato di $P$, che appunto è $C$, e si trasferisce il controllo all'inizio di $C$, dopo aver ovviamente legato i parametri formali con quelli attuali. Il significato di $P$ viene mantenuto nell'ambiente, cosicché alla successiva chiamata si possa recuperare $\rho(P) = C$.

Questa tecnica a volte viene chiamata *copy rule*, perché in effetti si copia il corpo $C$ della procedura $P$ al posto di $P$ stesso, tante volte quanto è necessario. Si noti che non si tratta di una macro-espansione, né si potrebbe macro-espandere per sempre la chiamata a $P$, perché non è noto a priori quante volte si debba chiamare la $P$ stessa. Per rendersene conto, si consideri di nuovo la funzione fattoriale: il suo corpo viene "usato" $n + 1$ volte se $n$ è l'argomento: di fatto si approssima la definizione della funzione dove la macro-espansione è ripetuta per sempre, avendo definito il suo comportamento sull'intervallo $[0..n]$, mentre per valori maggiori di $n$ il risultato è lasciato indefinito.

## 1.10 Problemi insolubili e riducibilità

In questo capitolo studieremo i problemi di appartenenza o di non appartenenza di un elemento ad un dato insieme, affrontando cosí il modo di risolvere problemi alternativo a quello considerato fino ad ora, in cui l'oggetto del discorso era calcolare funzioni. Ovviamente queste due visioni di un problema matematico sono strettamente correlate; in seguito vedremo *esattamente* come lo sono.

Una immediata corrispondenza si può facilmente stabilire correlando le funzioni con i loro domini,[21] per esempio

$$\lambda x.\, 2x \leftrightarrow \{\mathbb{N}\} \qquad\qquad \lambda x.\, x/2 \leftrightarrow \{2n \mid n \in \mathbb{N}\}$$

oppure si potrebbero correlare con le loro immagini, per esempio:

$$\lambda x.\, 2x \leftrightarrow \{2n \mid n \in \mathbb{N}\} \qquad\qquad \lambda x.\, x/2 \leftrightarrow \{\mathbb{N}\}$$

Relativamente agli esempi di sopra, si noti che ci sono infinite numerabili funzioni che hanno i naturali come dominio, quelle totali, e altrettante che hanno i naturali come immagine. Considerando ancora gli esempi di sopra vediamo che la funzione che calcola il doppio è totale e che la sua immagine sono i numeri pari, quindi non tutti i naturali (il viceversa per la funzione che calcola la metà). In ogni caso per questi esempi sia i domini che le immagini sono insiemi ricorsivi. Ricordiamone la definizione.

I è *ricorsivo* (ovvero *decidibile*) sse la sua funzione caratteristica

$$\chi_I(x) = \begin{cases} 1 & \text{se } x \in I \\ 0 & \text{se } x \notin I \end{cases}$$

è calcolabile totale.

Un esempio di insieme ricorsivo è costituito da quei numeri che soddisfano la condizione di Goldbach, ovvero quelli che sono somma di due numeri primi: $\{n = p + q \mid p, q \text{ primi}\}$. La funzione caratteristica di tale insieme può esser calcolata sommando tra loro tutti i primi minori o uguali a $n$, anche se si può far meglio ...

Vediamo che razza di insiemi si possono definire quando limitiamo il numero di passi nel calcolo di una MdT (e le dimensioni di ingressi e uscite).

**Esempio 1.10.1.** Siano dati $k, z \in \mathbb{N}$. Gli insiemi

$I = \{(i, x, k) \mid \exists y, n.\ (x, n < k)\ \wedge\ (M_i \text{ calcola } y = \varphi_i(x) \text{ in } n \text{ passi})\}$

$J = \{(i, x, k, z) \mid \exists n.\ (n < k) \wedge (M_i \text{ calcola } z = \varphi_i(x) \text{ in } n \text{ passi})\}$

[21] La correlazione è ancora piú evidente considerando l'insieme $\{(x, y) \mid \varphi(x) = y\}$.

sono entrambi ricorsivi. Infatti, la procedura calcolabile totale che decide $I$ è la seguente: fai andare la MdT $M_i$ su $x < k$ per al massimo $n - 1$ passi: se nel frattempo termina poni $\chi_I(i, x, k) = 1$, altrimenti poni $\chi_I(i, x, k) = 0$. Idem per la funzione caratteristica di $J$, controllando che il risultato sia $y$.

Si noti che $\varphi_i(x)\downarrow$ e restituisce $y$ in un numero $n$ di passi minore di un $k$ *qualsiasi*, e che quindi la seconda condizione limita il tempo accordato all'algoritmo per calcolare, mentre la prima invece ne limita anche lo spazio.

È interessante notare che la ricorsività degli insiemi appena considerati viene mantenuta anche se sostituissimo alla costante $k$ una funzione calcolabile totale $f$, che dipende dall'indice $i$ o dal dato $x$ o da entrambi. Vediamo allora quali insiemi vengono fuori se non poniamo alcun limite al numero dei passi consentiti a una macchina. Significa che andremo a vedere se una MdT termina (su un particolare dato) e se c'è un algoritmo per determinare tale proprietà – ovviamente no! Il gioco si fa definendo una classe di insiemi piú ampia di quella degli insiemi ricorsivi.

**Definizione 1.10.2.** Diciamo che un insieme $I$ è *ricorsivamente enumerabile* tutte le volte che è il dominio di una funzione calcolabile, formalmente

$$I \text{ è } \textit{ricorsivamente enumerabile} \text{ se e solamente se } \exists i.\, I = dom(\varphi_i).$$

Un insieme ricorsivamente enumerabile, detto anche *semi-decidibile* è quindi il dominio di una funzione calcolabile (il piú delle volte *parziale*, infatti se fosse totale $I = \mathbb{N}$), che è anche detta *semi-caratteristica* di $I$.

Ci sono ovvie relazioni tra gli insiemi ricorsivi (decidibili) e quelli ricorsivamente enumerabili (semi-decidibili).

**Proprietà 1.10.3.**

*i) se $I$ è ricorsivo allora $I$ è ricorsivamente enumerabile*

*ii) $I, \bar{I}$ sono ricorsivamente enumerabili se e solo se $I$ (e $\bar{I}$) sono ricorsivi.*

*Dimostrazione.* Il caso (i) ovvio: la $\varphi_i$ cercata restituisce 1 su $x$ se $\chi_I(x) = 1$, altrimenti diverge.
(ii) Il caso precedente basta per vedere la parte "*se*". Consideriamo allora la parte "*solo se*": siano $\varphi_i$ e $\varphi_{\bar{i}}$ le funzioni i cui domini sono rispettivamente $I$ e di $\bar{I}$. Adesso si ripeta il seguente ciclo: esegui un passo nel calcolo di $\varphi_i(x)$; se $\varphi_i(x) \downarrow$ allora $x \in I$ e poni $\chi_I(x) = 1$; altrimenti esegui un passo nel calcolo di $\varphi_{\bar{i}}(x)$; se $\varphi_{\bar{i}}(x) \downarrow$ allora $x \notin I$ e poni $\chi_I(x) = 0$. □

In realtà uno vorrebbe poter elencare (enumerare, generare) gli elementi di un insieme mediante una funzione calcolabile. Ecco un teorema che ci permette di fare ciò.

**Teorema 1.10.4.** *$I$ è ricorsivamente enumerabile se e solamente se è vuoto oppure è l'immagine di una funzione calcolabile* totale.

*Dimostrazione.* La parte *solo-se* è facile; quella piú complicata riguarda il caso in cui $I = dom(\varphi_i)$ sia non vuoto e consiste nella costruzione di una funzione totale calcolabile $f$ tale che $I = imm(f)$ a partire da $\varphi_i$. Innanzitutto, si cerca un elemento di $I$ mediante un procedimento a coda di colomba, come rappresentato dalla figura 1.2, in cui l'indice di riga $m$ rappresenta il numero dei passi del calcolo di $\varphi_i$ e l'indice di colonna $n$ il suo argomento. Piú precisamente, si eseguono $m$ passi nel calcolo di $\varphi_i(n)$, finché per qualche valore di $m$ e dell'argomento, sia $\bar{n}$, il calcolo si arresta; ovvero $\varphi_i(\bar{n}) \downarrow$ in $m$ passi e quindi $\bar{n} \in I$.

A questo punto, rappresentando con $\langle n, m\rangle$ il valore della codifica della coppia $(n, m)$, si inizia un secondo procedimento a coda di colomba eseguendo $\varphi_i(n)$ per $m$ passi: se tale calcolo si arresta, allora si pone $f(\langle n, m\rangle) = n$ (ovviamente $n \in dom(\varphi_i) = I$), altrimenti si pone $f(\langle n, m\rangle) = \bar{n}$ (per quanto detto prima $\bar{n} \in I$); si itera il procedimento incrementando la codifica $\langle n, m\rangle$, ovvero considerando $\langle n, m\rangle + 1$. Si generano cosí tutti gli elementi di $I$ con la funzione totale $f$. □

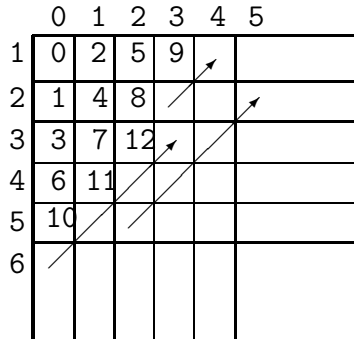



Figura 1.2: Nella tabella a "coda di colomba" si interpreti l'indice di riga come il numero di passi eseguiti da $M_i$ sul valore dell'indice di colonna.

Può essere interessante notare che dalla dimostrazione appena vista segue che è semi-decidibile verificare se una funzione calcolabile ha un dominio *non* vuoto.

Adesso vediamo un insieme veramente speciale e paradigmatico:

$$\boxed{K = \{x \mid \varphi_x(x) \downarrow\}}$$

**Proprietà 1.10.5.** *$K$ è ricorsivamente enumerabile.*

*Dimostrazione.* $K$ è il dominio di

$$\psi(x) = \begin{cases} x & \text{se } \varphi_x(x) \downarrow \\ \text{indefinita} & \text{altrimenti} \end{cases}$$

che è calcolabile: prendi la MdT $M_x$ e applicala a $x$; se e quando essa si arresta, restituisci $x$.

Se vi piace di più $K$ è il dominio di $\varphi_z(x, x)$ dove $\varphi_z$ è (uno degli indici de) l'algoritmo (o la macchina) universale del teorema d'enumerazione. □

Facciamo adesso vedere che la $\psi$ della dimostrazione precedente genera $K$, ma *non* è una funzione calcolabile totale, e che nessuna funzione che decida $K$ lo è.

**Proprietà 1.10.6.** *$K$ non è ricorsivo.*

*Dimostrazione.* Sia $\chi_K$ la funzione caratteristica di $K$ e per assurdo sia totale e calcolabile. Ma allora anche la seguente funzione

$$f(x) = \begin{cases} \varphi_x(x) + 1 & \text{se } x \in K \\ 0 & \text{se } x \notin K \end{cases}$$

sarebbe calcolabile totale. In questo modo otteniamo una contraddizione perché $\forall x.\ f(x) \neq \varphi_x(x)$; quindi non troviamo alcun indice per $f$ che di conseguenza non è calcolabile. □

La proprietà appena vista significa che *non esiste un algoritmo per decidere se $x \in K$* o no. Quindi questo problema è *insolubile*, anche se ovviamente è semi-decidibile. Inoltre *$\overline{K}$ non è ricorsivamente enumerabile*, quindi esistono problemi ancora più difficili di $K$! Infatti, se $\overline{K}$ fosse ricorsivamente enumerabile, sia $K$ che $\overline{K}$ sarebbero ricorsivi, per la proprietà 1.10.3(ii), in quanto $K$ è ricorsivamente enumerabile per la proprietà 1.10.5. Abbiamo cosí stabilito un piccolo frammento di gerarchia:

$$R \subsetneq RE \subsetneq nonRE$$

dove $R$ è la classe degli insiemi ricorsivi, $RE$ quella degli insiemi ricorsivamente enumerabili e $nonRE$ quella degli insiemi non ricorsivamente enumerabili (la scelta del nome "non ricorsivamente enumerabile" non è molto felice, perché un insieme (ricorsivo è anche) ricorsivamente enumerabile è anche "non ricorsivamente enumerabile:" in questo, come in mille altri casi, una formula è assai piú precisa di una frase nel linguaggio comune! con maggior esattezza si dovrebbe dire che $\overline{K} \in$ *co-RE*, la classe dei problemi i cui complementi sono ricorsivamente enumerabili, ma non ricorsivi), ecc.

Finalmente siamo arrivati al problema, che di solito si chiama *problema della fermata*, e che confuta l'affermazione di Hilbert che *tutti* i problemi matematici hanno una caratterizzazione esatta. A costo di essere noioso, val la pena di ripetere che, come tutti i risultati sulla calcolabilità che discuteremo, anche questo è *indipendente* sia dal formalismo impiegato per scrivere gli algoritmi, ovvero per esprimere le funzioni, sia dall'enumerazione effettiva scelta. In altre parole, *tutti* i formalismi che siano T-equivalenti soffrono del problema della fermata, il che potrebbe, con le dovute ipotesi, essere aggiunto come terzo punto al teorema di espressività.

Potrebbe comunque rimanere il dubbio che $K$ sia un problema artificiale, che non ha alcuna rilevanza pratica; si tratterebbe allora di un risultato negativo, ma di scarso impatto sulla realizzazione di elaboratori, programmi, sistemi e degli altri aggegi infernatici che ci stanno a cuore. Infatti a chi mai verrebbe in mente di applicare un programma a se stesso? A chi realizza compilatori, in particolare il cosidetto "bootstrapping"! Il problema può essere raccontato cosí: supponete di aver scritto un compilatore, o meglio un *cross-compiler*, in un certo linguaggio $L$, che traduce programmi scritti in $L$ in programmi scritti in un altro linguaggio $A$; rappresentiamo tale compilatore come $C_L^{L\to A}$ — questo ovviamente implica che abbiate già un compilatore per $L$ che gira su una qualche macchina, magari molto potente, ma su cui però i programmi in $A$ non girano. Supponete adesso di voler scrivere il compilatore $C_A^{L\to A}$, cioè un compilatore ancora da $L$ ad $A$, ma scritto stavolta nel linguaggio oggetto $A$, che potrebbe essere l'assembler di una macchina che non sostiene $L$. Basta allora prendere $C_L^{L\to A}$, applicarlo a sé stesso per ottenere ciò che si desidera:

$$C_L^{L\to A}(C_L^{L\to A}) = C_A^{L\to A}$$

Tuttavia questo può ancora sembrare un caso estremo, e consideriamo allora il seguente problema, la cui soluzione positiva ci aiuterebbe enormemente nel nostro lavoro. Possiamo scrivere un programma $P$ che, dato un altro programma $Q$ (individuato dal suo indice $y$) e un argomento $x$, ci assicura che la computazione di $Q$ su $x$ terminerà o meno? Questo è un problema certamente piú reale di $K$, prescinde dalla formalizzazione di algoritmo che stiamo esaminando e ha dunque interesse in sé. Infatti, piacerebbe a ciascuno di noi avere a disposizione il programma guardia $P$, in modo da non lanciare nemmeno l'esecuzione di $Q(x)$ quando questi non termina. Vista la sua importanza, questo problema si merita un nome:

**Problema della fermata**: dati $x, y.\ \varphi_y(x)\downarrow$ ? cioè $P_y(x)$ si ferma?

Il problema della fermata si formalizza e si studia nei termini usati in questo capitolo introducendo un altro insieme che gode di grande popolarità e caratterizzandone la natura. Sia

$$K_0 = \{(x, y) \mid \varphi_y(x) \downarrow\}, \text{ ovvero } K_0 = \{(x, y) \mid \exists z.\, T(y, x, z)\}.$$

dove $T$ è il il predicato di Kleene che abbiamo introdotto nel teorema di forma normale (1.9.3).

**Corollario 1.10.7.** *$K_0$ non è ricorsivo.*

*Dimostrazione.* Si ha che $x \in K$ se e solamente se $(x, x) \in K_0$, quindi se $K_0$ fosse ricorsivo lo sarebbe anche $K$ e viceversa. □

Abbiamo appena visto che il problema della fermata, formalizzato da $K_0$, è strettamente collegato al problema di decidere l'appartenenza o meno di un elemento a $K$, tanto da risultarne "equivalente." La tecnica di dimostrazione usata per collegare $K$ e $K_0$ si basa sul concetto di RIDUCIBILITÀ, che è una nozione fondamentale, e non solo nella teoria della calcolabilità e della complessità. Il punto cruciale nella sua definizione è che la *la funzione di riduzione* deve essere *"semplice"* (in questa parte del corso useremo funzioni calcolabili totali, nella terza parte funzioni polinomiali in tempo o indifferentemente logaritmiche in spazio).

Visto che la nozione di riducibilità è importantissima, faremo di seguito una digressione per introdurla e caratterizzarla in una forma abbastanza generale.

---

## RIDUCIBILITÀ

---

Una riduzione è una particolare funzione $f$ che trasforma un problema (ovvero un insieme o una classe) $A$ in un altro problema $B$, in modo da mantenerne inalterata la caratteristica principale.

**Definizione 1.10.8.** $A$ si *riduce* a $B$ secondo la *riduzione* $f$, in simboli $A \leqslant_f B$, tutte e sole le volte che

$a \in A$ se e solamente se $f(a) \in B$, ovvero $f(A) \subseteq B$ e $f(\bar{A}) \subseteq \overline{B}$

La seguente proprietà è di immediata dimostrazione.

**Proprietà 1.10.9.** *$A \leq_f B$ se e solamente se $\bar{A} \leq_f \overline{B}$*

*Dimostrazione.* Si ha che $x \in \bar{A}$ se e solamente se $x \notin A$ se e solamente se $f(x) \notin B$ se e solamente se $f(x) \in \overline{B}$. □

Piú in generale, si definisce una relazione di riduzioni ($\leqslant_F$) dove $F$ è una particolare classe di funzioni. Allora sciveremo

$$A \leqslant_F B \text{ se e solamente se esiste } f \in F \text{ tale che } A \leq_f B.$$

Ci interessano solo quelle riduzioni $\leqslant_F$ che danno origine a *classi* di problemi in qualche modo "omogenei." Vediamo adesso in maggior dettaglio cosa si debba intendere per omogeneo e come si possano allora definire quelle relazioni di riduzione che riteniamo interessanti.

**Definizione 1.10.10.** Siano $\mathcal{D}$ e $\mathcal{E}$ due classi di problemi con $\mathcal{D} \subseteq \mathcal{E}$ (e anche $\mathcal{H}$, con $\mathcal{E} \subseteq \mathcal{H}$, che però non menzioneremo ulteriormente). Una relazione di riduzione $\leqslant_F$ *classifica* $\mathcal{D}$ ed $\mathcal{E}$ se e solamente se per ogni problema $A, B, C$

i) $A \leqslant_F A$ *(Riflessiva)*

ii) $A \leqslant_F B, B \leqslant_F C$ implica $A \leqslant_F C$ *(Transitiva)*

iii) $A \leqslant_F B, B \in \mathcal{D}$ implica $A \in \mathcal{D}$ *($\mathcal{D}$ ideale = chiuso all'ingiú per riduzione)*

iv) $A \leqslant_F B, B \in \mathcal{E}$ implica $A \in \mathcal{E}$ *($\mathcal{E}$ ideale = chiuso all'ingiú per riduzione)*

Vediamo adesso una caratterizzazione differente, ma del tutto equivalente, delle riduzioni che classificano coppie di classi, l'una inclusa nell'altra.

**Lemma 1.10.11.** *Una relazione di riduzione $\leqslant_F$ classifica $\mathcal{D}$ ed $\mathcal{E}$, tali che $\mathcal{D} \subseteq \mathcal{E}$, se*

*i)* $id \in F$ *($F$ ha identità)*

*ii)* $f, g \in F \Rightarrow f \circ g \in F$ *($F$ chiusa per composizione)*

*iii)* $f \in F,\ B \in \mathcal{D} \Rightarrow \{x \mid f(x) \in B\} \in \mathcal{D}$

*iv)* $f \in F,\ B \in \mathcal{E} \Rightarrow \{x \mid f(x) \in B\} \in \mathcal{E}$

*Dimostrazione.* Vedi punto per punto la definizione di classificazione. □

Attraverso il concetto di relazione di riduzione che classifica due classi di problemi si possono definire le seguenti nozioni molto importanti. (Si noti che la relazione $\leqslant_F$ è un pre-ordine parziale,[22] il che giustifica l'uso del termine "ideale" fatto sopra.)

[22] Cioè un ordinamento parziale riflessivo e transitivo, ma non anti-simmetrico.

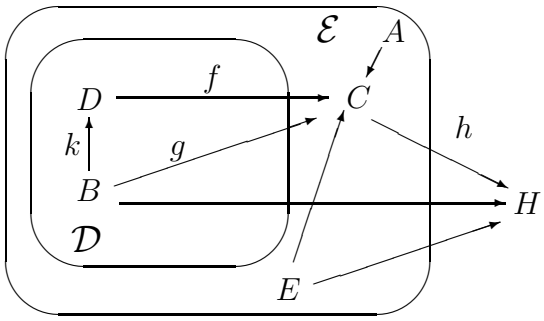
**Definizione 1.10.12.** Se $\leqslant_F$ classifica $\mathcal{D}$ ed $\mathcal{E}$, $\forall A, B, H$ problemi

i) $A \equiv B$ se $A \leqslant_F B$ e $B \leqslant_F A$
(si dice anche che $\{B \mid A \equiv_F B\}$ è il *grado* di A, o anche che $A$ è equivalente a $B$ rispetto a $\leqslant_F$)[23]

ii) $H$ è $\leqslant_F$-*arduo* per $\mathcal{E}$ se $\forall A \in \mathcal{E}.\ A \leqslant_F H$ [24]

iii) $H$ è $\leqslant_F$-*completo* per $\mathcal{E}$ se $H$ è $\leqslant_F$-arduo per $\mathcal{E}$ e $H \in \mathcal{E}$

Diremo semplicemente $\mathcal{E}$-arduo o $\mathcal{E}$-completo quando la classe di riduzioni $F$ sia fissata; talvolta ometteremo anche $\mathcal{E}$, se chiaro dal contesto.




Il disegno di sopra esemplifica quanto scritto. Il problema $C$ è completo per $\mathcal{E}$ e a esso si riducono sia $B$ e $D$ di $\mathcal{D}$, sia $A$ e $E$ di $\mathcal{E}$; non tutte le riduzioni sono state disegnate (come frecce), tuttavia si noti che $g = f \circ k$ e allo stesso modo si compongono tutte le frecce disegnate o meno; infine $H$ è un problema arduo per $\mathcal{E}$, ma non completo: tutti i problemi di $\mathcal{E}$ si riducono ad $H$, ma $H \notin \mathcal{E}$.

Se un problema è completo per una classe $\mathcal{E}$ e appartiene ad una sotto-classe $\mathcal{D}$, allora le due classi coincidono.

**Proprietà 1.10.13.** *Se $\leqslant_F$ classifica $\mathcal{D}$ ed $\mathcal{E}$, $\mathcal{D} \subseteq \mathcal{E}$ e $C$ è completo per $\mathcal{E}$, allora $C \in \mathcal{D}$ se e solamente se $\mathcal{D} = \mathcal{E}$.*

*Dimostrazione.* (se) ovvia.
(solo se) Sia $C \in \mathcal{D}$ e $A \in \mathcal{E}$. Per completezza $A \leqslant_F C$ e $A \in \mathcal{D}$ per la condizione (iii) di $\leqslant_F$ che classifica $\mathcal{D}$ ed $\mathcal{E}$. Quindi $\mathcal{E} \subseteq \mathcal{D}$ e la tesi. □

[23] Se consideriamo, p.e. $\mathcal{D}_{\equiv}$ allora $\leqslant_F$ diventa un ordinamento parziale.
[24] Potrebbe essere $H \in \mathcal{H} \setminus \mathcal{E}$.

Inoltre è facile capire quali siano gli elementi del grado di $A$, problema $\leqslant_F$-completo per $\mathcal{E}$. Per aiutare l'intuizione guardiamo nuovamente il disegno fatto sopra: se vi fosse una riduzione anche tra $C$ e $A$, allora componendo le frecce si otterebbe una riduzione tra ogni elemento di $\mathcal{E}$ e $A$, cioè anche $A$ sarebbe completo per $\mathcal{E}$.

**Proprietà 1.10.14.** *Se $\leqslant_F$ classifica $\mathcal{D}$ ed $\mathcal{E}$, $\mathcal{D} \subseteq \mathcal{E}$ e $A$ è completo per $\mathcal{E}$, $A \leqslant_F B$, e $B \in \mathcal{E}$, allora $B$ è completo per $\mathcal{E}$.*

*Dimostrazione.* Per ogni $D \in \mathcal{E}$, $D \leqslant_F A$ per completezza, ma $\leqslant_F$-classifica $\mathcal{D}$ ed $\mathcal{E}$ e allora $D \leqslant_F A$ e $A \leqslant_F B$ implicano $D \leqslant_F B$ e quindi $B$ è arduo e, poiché appartiene a $\mathcal{E}$, è completo. □

Un problema completo per $\mathcal{E}$ gioca un ruolo rilevantissimo, in quanto "rappresenta la difficoltà" massima dei problemi di $\mathcal{E}$. Infatti, è facile vedere che il grado di un problema $A$ completo per $\mathcal{E}$ è il grado massimo di $\mathcal{E}$ in $\leqslant_F$. Inoltre valgono le seguenti affermazioni (anche per problemi non completi, a dire il vero):

- se $B \leqslant_F A$ allora $B$ *al piú* ha il (o meglio al piú appartiene al) grado di $A$, cioè è piú facile o altrettanto difficile di $A$;
- se $A \leqslant_F B$ allora $B$ ha *almeno* il grado di $A$, cioè è di difficoltà maggiore o uguale a quella di $A$.

FINE DIGRESSIONE

Rifrasiamo ora le definizioni 1.10.8 e 1.10.10 per ottenere il concetto di riducibilità che useremo in questa parte del corso; per farlo diamo un nome alla classe delle funzioni calcolabili totali:
$rec = \{\varphi_x \mid \forall y \in \mathbb{N}.\, \varphi_x(y) \downarrow\}$.

**Definizione 1.10.15.** $A$ è riducibile a $B$ ($A \leq_{rec} B$) se e solamente se esiste una funzione calcolabile totale $f : \mathbb{N} \to \mathbb{N}$ tale che $x \in A$ se e solamente se $f(x) \in B$.

Vediamo ora che queste relazioni di riduzione *conservano la ricorsività e la ricorsiva enumerabilità*. Come già fatto quando abbiamo introdotto il piccolo frammento di gerarchia, d'ora in avanti indichiamo con $R$ e $RE$ le classi di insiemi rispettivamente ricorsivi e ricorsivamente enumerabili. Allora, possiamo dimostrare quanto segue.

**Teorema 1.10.16.** *La relazione di riduzione $\leq_{rec}$ classifica $R$ ed $RE$.*

*Dimostrazione.* Sappiamo già che $R \subseteq RE$ grazie alla proprietà 1.10.3(i). Possiamo allora usare il lemma 1.10.11 per dimostrare la tesi. Facciamo allora vedere che tutte le ipotesi del lemma sono soddisfatte:

i) Facile, dalla definizione di funzione $\mu$-ricorsiva.

ii) Ovvio perché la composizione conserva la totalità.

iii) La funzione caratteristica di $\{x \mid f(x) \in B\}$ è $\chi_B \circ f$, che è calcolabile totale perché $f$ e $\chi_B$ sono entrambe calcolabili totali.

iv) Analoga al punto precedente, con la semi-caratteristica di $B$. □

Nei teoremi e negli esercizi che seguono useremo quasi sempre funzioni di riduzione iniettive, di solito ottenute applicando il teorema del parametro (teorema 1.9.12); piú in generale si potrebbero definire relazioni di riduzioni $\leq^m_{rec}$, in cui le funzioni (calcolabili totali) di riduzione usate non sono iniettive (cioè sono molti-a-uno nella terminologia nord-americana).

Il fatto che $\leq_{rec}$ classifichi $R$ ed $RE$ può essere intuitivamente visto come la capacità che le riduzioni con funzioni calcolabili totali hanno di *separare* i problemi ricorsivi da quelli ricorsivamente enumerabili. Ciò vien fatto giocando sul tempo necessario a decidere un problema: se questo è ricorsivo avremo la risposta in tempo *finito*, altrimenti il tempo necessario è *infinito*. Inoltre, ci basta trovare un problema che sia $\leq_{rec}$-completo per $R$ per poter vedere quali problemi sono decidibili e quali no; ancora piú interessante è trovare un problema che sia $\leq_{rec}$-completo per $RE$: sapremmo allora quali problemi sono (al piú) semi-decidibili e quali nemmeno semi-decidibili. Infatti basta ridurre il problema da studiare a quello completo e sapremo che è ricorsivamente enumerabile oppure ridurre il problema completo a quello da studiare e sapremo che quest'ultimo, ben che ci vada, è ricorsivamente enumerabile. Infatti, come notato nella digressione:

- Se $A \leqslant_{rec} B$ e $B$ è ricorsivamente enumerabile ($B \in RE$), allora $A$ è ricorsivamente enumerabile (e forse anche ricorsivo).
- Se $A \leqslant_{rec} B$ e $A$ non è ricorsivamente enumerabile ($A \notin RE$), allora $B$ non è ricorsivamente enumerabile (e men che meno ricorsivo).

Inoltre, se $A$ è ricorsivo il fatto che esso si riduca a $B$ non ci consente di dedurre alcunché sulla natura di $B$, il quale potrebbe essere ricorsivo o ricorsivamente enumerabile o nemmeno ricorsivamente enumerabile; analogamente nel caso di $A$ ricorsivamente enumerabile.

Prima di vedere il nostro (primo) problema completo per $RE$, notiamo che né $K$ si riduce con funzioni calcolabili totali a $\overline{K}$ né il viceversa, cioè vediamo che un insieme ricorsivamente enumerabile può essere inconfrontabile con uno non ricorsivamente enumerabile, usando $\leq_{rec}$ (si ricordi la definizione di $K$ a pagina 60). (Questo può apparire bizzarro, perchè se scopro che $x \in K$ so anche che $x \notin \overline{K}$ e infatti ci sono riduzioni un po' piú astute che permettono di confrontare anche $K$ e $\overline{K}$.) Dobbiamo quindi mostrare che

$$\overline{K} \not\leq_{rec} K \qquad \text{e} \qquad K \not\leq_{rec} \overline{K}.$$

La prima diseguaglianza deve essere vera, perché altrimenti, usando la proprietà 1.10.3(ii), $\overline{K}$ sarebbe ricorsivamente enumerabile e anche ricorsivo cosí come $K$, poiché quest'ultimo è ricorsivamente enumerabile per la proprietà 1.10.6; per provare la seconda diseguaglianza basta considerare che $A \leq_{rec} B$ se e solamente se $\overline{A} \leq_{rec} \overline{B}$ (cf. la proprietà 1.10.9). Quindi se valesse $K \leq_{rec} \overline{K}$ avremmo anche $\overline{K} \leq_{rec} K$, che è appena stato dimostrato falso.

Naturalmente l'insieme $RE$- completo non può che essere $K$!

**Teorema 1.10.17.** *$K$ è $RE$-completo, ovvero $\leq_{rec}$-completo per $RE$.*

*Dimostrazione.* Dobbiamo dimostrare che se $A \in RE$ allora $A \leq_{rec} K$. Per definizione $A$ è il dominio di una funzione calcolabile $\psi$, cioè $A = \{x \mid \psi(x)\downarrow\}$. A partire da $\psi$ si definisca una funzione $\psi'$ a due variabili di cui ignora la seconda, cioè sia $\psi'(x,y) = \psi(x)$, che è a sua volta una funzione calcolabile e quindi avrà un indice, diciamo $i$; in simboli $\psi' = \varphi_i$. Allora, per il teorema del parametro $\psi'(x,y) = \varphi_i(x,y) = \varphi_{s(i,x)}(y)$, con $s$ calcolabile, iniettiva e totale. Posso riscrivere la definizione di $A$ come segue:

$$\begin{aligned}
A &= \{x \mid \psi(x)\downarrow\} \\
&= \{x \mid \psi'(x,y)\downarrow\} \\
&= \{x \mid \varphi_i(x,y)\downarrow\} \\
&= \{x \mid \varphi_{s(i,x)}(y)\downarrow\} \text{ per il teorema del parametro} \\
&= \{x \mid \varphi_{s(i,x)}(s(i,x))\downarrow\} \text{ ponendo } y = s(i,x) \\
&= \{x \mid s(i,x) \in K\}
\end{aligned}$$

quindi $x \in A$ se e solamente se $f(x) \in K$, con $f(x) = \lambda x.s(i,x)$, che è totale, calcolabile e iniettiva perchè la $s(i,x)$ lo è (si veda la nota a pié di pagina 56: prendiamo $i$ tale che $\psi'(x,y) = \varphi_i(x,y) = \varphi_{s(i,x)}(y)$ da cui $f = \lambda x.\ s(i,x)$). □

**Esercizio di riduzione**

Vediamo il seguente esercizio, affatto banale, con cui si mostra che l'insieme degli indici delle funzioni calcolabili totali $TOT$ non è ricorsivo, quindi non si può decidere in tempo finito se una funzione calcolabile è totale. Dimostriamo che

$$K = \{x \mid \varphi_x(x)\downarrow\} \leq_{rec} \{x \mid \varphi_x \in rec\} = TOT \text{ [25]}$$

Definiamo ora questa funzione: $\psi(x,y) = \begin{cases} 1 & \text{se } x \in K \\ \text{indef} & \text{altrimenti} \end{cases}$

La nostra $\psi$ è calcolabile parziale: il programma $P_x$ calcola $\varphi_x(x)$ e se e quando questa converge, restituisce 1 per ogni $y$. Per il teorema *s-m-n* esiste $f$ calcolabile totale iniettiva tale che $\varphi_{f(x)}(y) = \psi(x,y)$. (Per costruire la $f$ si ricordi la nota a pié di pagina 56: si scelga $i$ tale che $\psi(x,y) = \varphi_i(x,y) = \varphi_{s(i,x)}(y)$ da cui $f = \lambda x.\ s(i,x)$). Adesso

$$x \in K \Rightarrow \varphi_{f(x)} = \psi(x,y) = \lambda y.\ 1 \Rightarrow \varphi_{f(x)} \text{ totale} \Rightarrow f(x) \in TOT$$

$$x \notin K \Rightarrow \varphi_{f(x)} = \lambda y.\ \text{indefinito} \Rightarrow \varphi_{f(x)} \text{ non è totale} \Rightarrow f(x) \notin TOT.$$

Di conseguenza, $TOT$ è *ben che ci vada* ricorsivamente enumerabile.

Nell'esercizio appena svolto abbiamo usato un insieme che contiene *tutti e soli* gli indici delle funzioni calcolabili che hanno la proprietà di essere totali. La stessa cosa può essere fatta considerando altre proprietà delle funzioni. Per esempio, si potrebbe definire l'insieme $A$ di *tutti e soli* gli indici dei programmi che calcolano una particolare funzione $\varphi$ (si confronti questo insieme $A$ con $A_x$, definito nel teorema 1.9.2 che contiene *solo* indici della stessa funzione, ma *non tutti*); definito un tale insieme $A$, ci si può riferire indifferentemente ad esso oppure a $\varphi$ in quanto descrivono entrambi la *medesima* entità. Piú formalmente abbiamo la seguente definizione.

**Definizione 1.10.18.** $A$ è un *insieme di indici che rappresentano le funzioni* se e solamente se

$$\forall x, y.\ \text{se } x \in A \text{ e } \varphi_x = \varphi_y \text{ allora } y \in A.$$

Adesso passiamo a studiare quegli insiemi di indici $A$ che rappresentano le funzioni e tali per cui $K \leq_{rec} A$ (oppure $K \leq_{rec} \overline{A}$). Così sapremo quali classi di funzioni sono al massimo semi-decidibili (ma non decidibili), perché, come detto sopra un insieme di indici $A$ che rappresentano le funzioni *individua esattamente* le funzioni calcolate dalle macchine che hanno indice in $A$.

[25] Non si confondano $TOT$ e $rec$: il primo è un insieme di *indici*, cioè di macchine, programmi; il secondo è un insieme di *funzioni*.

**Teorema 1.10.19.** *Sia $A$ un insieme di indici che rappresentano le funzioni tale che $\emptyset \neq A \neq \mathbb{N}$. Allora $K \leq_{rec} A$ oppure $K \leq_{rec} \overline{A}$.*

*Dimostrazione.* Prendi $i_0$ tale che $\varphi_{i_0}(y)$ sia ovunque indefinita. Supponiamo che $i_0 \in \overline{A}$ e dimostriamo $K \leq_{rec} A$ (se $i_0 \in A$ si procede in modo simmetrico). Poichè $A \neq \emptyset$ scegli $i_1 \in A$. Hai $\varphi_{i_0} \neq \varphi_{i_1}$ perché $A$ è un insieme di indici che rappresentano le funzioni. Definiamo adesso la seguente funzione che è calcolabile:

$$\psi(x,y) = \varphi_{f(x)}(y) = \begin{cases} \varphi_{i_1}(y) & \text{se } x \in K \\ \text{indefinita } = \varphi_{i_0}(y) & \text{altrimenti} \end{cases}$$

dove, usando il teorema *s-m-n*, abbiamo determinato la $f$ funzione calcolabile totale iniettiva (come suggerito nella nota a pié di pagina 56: sia $i$ tale che $\psi(x,y) = \varphi_i(x,y) = \varphi_{s(i,x)}(y)$, allora si pone $f = \lambda x.\ s(i,x)$). Allora

$$x \in K \text{ implica } \varphi_{f(x)} = \varphi_{i_1} \text{ implica } f(x) \in A$$

perchè $i_1 \in A$ e $A$ è un insieme di indici che rappresentano le funzioni e quindi anche $f(x) \in A$. Viceversa, dato che $i_0 \in \overline{A}$,

$$x \notin K \text{ implica } \varphi_{f(x)} = \varphi_{i_0} \text{ implica } f(x) \in \overline{A} \text{ (implica } f(x) \notin A).$$

□

**Nota**. Esistono insiemi $B$ (per esempio TOT) tali che $K \leq_{rec} B$ e *anche* $K \leq_{rec} \overline{B}$, cioè esistono $f$ e $g$ calcolabili totali iniettive tali che $x \in K$ se e solo se $f(x) \in B$ e $x \in K$ se e solo se $g(x) \in \overline{B}$.

A titolo di curiosità e completezza, facciamo vedere che $K \leq_{rec} \overline{TOT}$, definendo la seguente funzione che è calcolabile perché il controllo di convergenza è fatto su un numero *finito* di passi.

$$\varphi_{f(x)}(y) = \psi(x,y) = \begin{cases} 1 & \text{se } \varphi_x(x) \text{ non converge entro } y \text{ passi} \\ \text{indefinita} & \text{altrimenti} \end{cases}$$

Abbiamo allora che:

- $x \in K \Rightarrow \exists \bar{y}.\varphi_x(x)\downarrow$ in $\bar{y}$ passi $\Rightarrow \varphi_{f(x)}(y) = \begin{cases} 1 & \text{se } y < \bar{y} \\ \text{indefinita} & \text{altrimenti} \end{cases} \Rightarrow$ $f(x) \in \overline{TOT}$

e che

- $x \notin K \Rightarrow \nexists \bar{y}.\varphi_x(x)\downarrow$ in $\bar{y}$ passi $\Rightarrow \forall y\, \varphi_{f(x)}(y) = 1 \Rightarrow f(x) \in TOT$

Il seguente corollario del teorema precedente è di immediata dimostrazione è di particolare importanza, tanto esser chiamato teorema, perché pone dei limiti drastici alle proprietà dimostrabili sulle funzioni calcolabili.

**Teorema 1.10.20** (Rice)**.** *Sia $\mathcal{A}$ una classe di funzioni calcolabili. L'insieme $A = \{n \mid \varphi_n \in \mathcal{A}\}$ è ricorsivo se e solo se $\mathcal{A} = \emptyset$ oppure $\mathcal{A}$ è la classe di* tutte *le funzioni calcolabili.*

*Dimostrazione.* Si noti che $A$ è un insieme di *indici* mentre $\mathcal{A}$ è una classe di *funzioni* (anche se la lettera è la stessa, il carattere è *diverso*! a indicare che i primi sono *sintassi*, mentre i secondi sono *semantica*).
La dimostrazione è immediata per i casi banali, cioè quando $\mathcal{A} = \emptyset$ e quando $\mathcal{A}$ è la classe di *tutte* le funzioni calcolabili.
Negli altri casi, basta applicare il teorema 1.10.19, poiché $A$ è un insieme di indici che rappresentano le funzioni, il quale non è vuoto, perché $\mathcal{A}$ contiene almeno una funzione, né coincide con $\mathbb{N}$, perché $\mathcal{A}$ non contiene tutte le funzioni calcolabili. □

Questo importante risultato negativo ovviamente si ripercuote sulle proprietà che si possono dimostrare sui programmi: ogni metodo di prova si scontra inevitabilmente con il problema della fermata. Gli informatici però non si sono arresi e hanno sviluppato svariate tecniche per aggirare il problema. Una famiglia di analizzatori di programmi particolarmente usata è quella che va sotto il nome di *analisi statica* [vedi Hankin, Nilson, Riis Nilson]. In breve, il *testo* del programma viene scrutinato e si raccolgono informazioni su come gli oggetti che vi compaiono (per esempio variabili, chiamate di procedura, ecc.) verrano usati a tempo di esecuzione (per esempio se i valori che verranno assegnati alle variabili sono del giusto tipo, se prima di usarle avranno ricevuto un valore di inizializzazione, ecc.). Il gioco ha successo perché il comportamento dei programmi viene *approssimato* in modo sicuro, ovvero ciò che viene predetto è una sovra-approssimazione di ciò che succederà davvero a tempo di esecuzione (per esempio potrà succedere di dire che tra i valori che si potrebbero assegnare a una variabile intera c'è una stringa senza che questo avvenga davvero a tempo di esecuzione, ma *non* capiterà mai di dire che tutti i valori sono interi se a tempo di esecuzione a tale variabile viene assegnata una stringa). In altre parole si dice che si può sbagliare rimanendo sul lato giusto, ovvero senza conseguenze, perché si afferma che un programma corretto non lo è, ma mai si afferma che in programma scorretto è invece corretto. A questa famiglia di analizzatori appartengono vari strumenti spesso incorporati nei compilatori, da cui il nome statici, tra cui i *type-checker*, gli analizzatori *data-flow* o *control-flow* e molti altri ancora.
Un'applicazione immediata del teorema di Rice è che:

$$K_1 = \{x \mid \text{dominio}(\varphi_x) \neq \emptyset\}$$

cioè l'insieme (degli indici) delle funzioni che sono definite in almeno un punto *non è ricorsivo*, sebbene sia ricorsivamente enumerabile. Inoltre si

può dimostrare facilmente che $K \equiv K_0 \equiv K_1$, cioè i tre insiemi si riducono l'uno all'altro e sono $RE$-completi.

Altre classi che non sono ricorsive sono

$$\text{FIN} = \{x \mid \text{dominio}(\varphi_x) \text{ finito}\}$$

$$\text{INF} = \{x \mid \text{dominio}(\varphi_x) \text{ è infinito}\} = \mathbb{N} \backslash \text{FIN}$$

$$\text{TOT} = \{x \mid \varphi_x \text{ totale}\} = \{x \mid \text{dominio}(\varphi_x) = \mathbb{N}\}$$

$$\text{REC} = \{x \mid \text{dominio}(\varphi_x) \text{ è ricorsivo}\}$$

$$\text{CONST} = \{x \mid \varphi_x \text{ totale e costante}\}$$

$$\text{EXT} = \{x \mid \varphi_x \text{ è estendibile a funzione calcolabile totale}\}$$

Si noti che trova qui la sua giustificazione l'affermazione fatta a suo tempo che non esiste un algoritmo che termini sempre (e nemmeno una funzione di semi-decisione, per quanto detto subito sotto) per trovare tutte le funzioni totali o tutte quelle estendibili.

Inoltre si può vedere che gli insiemi listati sopra *non sono nemmeno* ricorsivamente enumerabili. Infatti, si può dimostrare che

$$\overline{K} \leq_{rec} \text{FIN}, \ldots, \text{TOT}, \ldots, \text{EXT}.$$

Visto che $\overline{K}$ non è ricorsivamente enumerabile (altrimenti sarebbe ricorsivo così come lo sarebbe $K$), ne segue immediatamente che questi problemi non si possono nemmeno semi-decidere!

Concludiamo con un paio di osservazioni ovvie. Il complemento di un insieme non ricorsivamente enumerabile può essere a sua volta non ricorsivamente enumerabile, come mostrato da FIN e dal suo complemento INF; si ricordi che ciò non è vero per gli insiemi ricorsivamente enumerabili, a meno che non siano anche ricorsivi. Inoltre, tra i sottoinsiemi di un insieme non ricorsivamente enumerabile (per esempio INF) ve ne sono sia di non ricorsivamente enumerabili (per esempio REC), che di ricorsivamente enumerabili (per esempio $K$), che di ricorsivi (per esempio $\emptyset$) — magari ricorsivi ma non infiniti: è il caso dei cosí detti insiemi *immuni*. Quanto detto è banalmente vero anche per gli insiemi ricorsivamente enumerabili e per quelli ricorsivi, primo fra tutti $\mathbb{N}$.

**Nota Bene**. In questa pagina ci sono due errori ...[26]

[26] Alessandro e Jacopo nel 2020, Filippo nel 2021 e Pietro nel 2023 hanno trovato lo stesso errore, ma l'altro dov'è? :-) — grazie!