Appunti di

Fondamenti di Informatica

Rosario Terranova

v 1.1.4

http://rosarioterranova.blogspot.it/

Elementi di Teoria dei linguaggi formali

I linguaggi formali sono delle strutture matematiche che svolgono un ruolo particolarmente importante nell'ambito informatico.

Alfabeto Stringhe e linguaggi

Alfabeto

Indicato con Σ, è un insieme finito non vuoto di simboli detti caratteri.

Es. L'insieme delle lettere dell'alfabeto italiano è rappresentabile con $\Sigma = \{a, b, c, d, \dots, z\}$.

Es. alfabeto del linguaggio C = {a,z,0,9,+,-,<,>,%,!,#,…}

Stringhe

Dato un alfabeto Σ, una stringa (o parola) è una sequenza finita di caratteri denotata come $< \Sigma^*, \circ, \varepsilon >$, ovvero *monoide sintattico* che definisce:

- Σ^* come l'insieme delle *parole* o *stringhe*;
- ε la *parola vuota;*
- l'operazione $\circ$ la *concatenazione*, che consiste nel giustapporre due parole di Σ^*.

Es. Dato $\Sigma^* = \{a, b, c, \dots\}$ e $\Sigma^* = \{1,2,3, \dots\}$, $\Sigma^* \circ \Sigma^* = \{a1, b2, c3, \dots\}$.

Vale naturalmente la proprietà: $\forall x, x \circ \varepsilon = \varepsilon \circ x = x$.

Con la notazione |x| indichiamo la *lunghezza* di una parola x, ovvero il numero di caratteri che la costituiscono. Chiaramente $|\varepsilon| = 0$.

La concatenazione non gode della proprietà commutativa $x \circ y \neq y \circ x$, e inoltre la stringa può essere concatenata con se stessa quando indicata con x^h(la stringa x è concatenata con se stessa h volte).

Es. |CIAO| = 4

Linguaggio

Si definisce linguaggio un qualsivoglia sottoinsieme di Σ^*. Dato che $\Sigma \subseteq \Sigma^*$, un alfabeto è a sua volta un linguaggio.

Con Λ si indica il linguaggio che non contiene alcuna stringa. $\Lambda \neq \{\varepsilon\}$.

Es. $\Sigma = \{a, b\}$, l'insieme $\{a^n, b^n | n \geq 1\}$ è il linguaggio composto da tutte le stringhe costituite dalla concatenazione di un certo numero di a, seguita dalla concatenazione dello stesso numero di b. $aaabbb \in L$, $aabbb \notin L$.

Con Σ^+ denotiamo l'insieme delle stringhe non vuote. Pertanto $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$

Operazioni sui linguaggi

Dati due linguaggi L_1 e L_2 si possono definire su essi varie operazioni:

Operazione	Definizione	Es. Dati $L_1 = \{aa, bb\}$ e $L_2 = \{bb, ab\}$
Intersezione	$L_1 \cap L_2 = \{x \in \Sigma^* \mid x \in L_1 \wedge x \in L_2\}$	$L_1 \cap L_2 = \{bb\}$
Unione	$L_1 \cap L_2 = \{x \in \Sigma^* \mid x \in L_1 \vee x \in L_2\}$	$L_1 \cup L_2 = \{aa, bb, ab\}$
Complemento	$\overline{L_1} = \Sigma^* - L_1; \overline{L_1} = \{x \in \Sigma^* \mid x \notin L_1\}$	$\overline{L_1 \cap L_2} = \{aa, ab\}$
Concatenazione	$L_1 \circ L_2 = \{x \in \Sigma^* \mid \exists y_1 \in L_1 \wedge \exists y_2 \in L_2 (x = y_1 \circ y_2)\}.$	$L_1 \circ L_2 = \{aabb, aaab, bbbb, bbab\}$
Potenza	$L^h = L \circ L^{h-1}, h \geq 1$	$(L_1)^2 = \{aaaa, aabb, bbbb, bbaa\}$
Iterazione	$L^* = \bigcup_{h=0}^{\infty} L^h$ L'operatore * prende il nomedi iterazione o stella di Kleene.	Dati i linguaggi $L_1 = \{BIS\}$ $L_2 = \{NONNO\}$, il linguaggio $L_1^* \circ L_2$ contiene NONNO, BISNONNO, BISBISNONNO…
Iterazione senza 0	$L^+ = \bigcup_{h=1}^{\infty} L^h$	

Cardinalità dei linguaggi

Dato un alfabeto $\Sigma = \{a_1, \dots, a_n\}$ possiamo stabilire un ordinamento lessicografico delle stinghe di Σ^* l'ordinamento ottenuto stabilendo un qualunque ordinamento tra i caratteri di Σ ($a_1 < a_2 < \cdots < a_n$), e definendo l'ordinamento di due stringhe $x, y \in \Sigma^*$ tale che $x < y$, ma anche $|x| < |y|$.

Es. Dato $\Sigma = \{a, b\}$ dove assumiamo $a < b$, le stringhe Σ^* sono enumerate come segue: $\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, \dots$

Linguaggi formali

Un problema basilare dell'informatica è riconoscere se una stringa appartenga o meno ad un determinato linguaggio, dato che non sempre esiste un algoritmo che riconosca la stringa tra i numerosi linguaggi esistenti.

Nell'ambito informatico siamo interessati a linguaggi generalmente infiniti, le cui stringhe sono caratterizzate da una particolare proprietà: ad esempio il linguaggio dei programmi C è costituito da tutte le stringhe che soddisfano le proprietà di poter essere analizzate con un compilatore C senza che venga rilevato alcun errore sintattico. Lo studio formale dei linguaggi si occupa dei metodi di definizione della sintassi di un linguaggio di programmazione, dei metodi per la verifica che un programma soddisfi le proprietà sintattiche volute e dei metodi di traduzione da un linguaggio ad un altro. La definizione di linguaggi si può effettuare mediante strumenti diversi:

- Espressioni regolari: consentono di definire linguaggi di tipo particolare, con una struttura piuttosto semplice. Tuttavia non sono idonee a rappresentare linguaggi più complessi, come ad esempio i linguaggi di programmazione.
- Approccio generativo: si utilizzano opportuni strumenti formali, le *grammatiche formali* appunto, che consentono di costruire le stringhe di un linguaggio tramite un insieme prefissato di regole, dette *regole di produzione*. Non consente tuttavia di impostare in modo algoritmo il problema del riconoscimento di un linguaggio, cioè il problema di decidere se una stringa x ∈ al linguaggio L.
- Approccio riconoscitivo: consiste nell'utilizzare macchine astratte, dette *automi riconoscitori*, che definiscono algoritmi di riconoscimento dei linguaggi stessi.

Espressioni regolari

Le espressioni regolari sono un metodo formale (o algebrico) di rappresentazione di un linguaggio; tale rappresentazione avviene mediante una opportuna interpretazione dei simboli che la compongono.

Es. se l'espressione e_1 rappresenta il linguaggio L_1 e l'espressione e_2 rappresenta il linguaggio L_2, il linguaggio $L_1 \cup L_2$ è rappresentato dall'espressione $e_1 + e_2$

Dato un alfabeto Σ, chiamiamo espressione regolare una stringa r sull'alfabeto Σ che utilizza l'insieme di simboli $\{+, *, (,), \cdot, \emptyset\}$ tale che:

1. $r = \emptyset$
2. $r \in \Sigma$
3. $r = (s + t) \vee r = (s \cdot t) \vee r = s^*$ dove s e t sono espressioni regolari su Σ

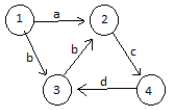
Espr. regolari	Linguaggi
r = ∅	L(r) = ∧
r = a	L(r) = $\{a\}$
r = $(s+t)$	L(r) = $\mathcal{L}(s) \cup \mathcal{L}(t)$
r = $(s \cdot t)$	L(r) = $\mathcal{L}(s) \circ \mathcal{L}(t)$
r = s^* (ripetizioni)	L(r) = $\big(\mathcal{L}(s)\big)^*$

Es. L'espressione regolare $(a+b)^*a$ rappresenta il linguaggio dove
$L((a+b)^*a) =$
$= \mathcal{L}((a+b)^*) \circ \mathcal{L}(a)$
$= (\mathcal{L}(a+b)^*) \circ \mathcal{L}(a)$
$= (\mathcal{L}(a) \cup (b)^*) \circ \{a\}$
$= (\{a\} \cup \{b\})^* \circ \{a\}$
$= \{a,b\}^* \circ \{a\} \quad = \{x | x \in \{a,b\}^+, x\ termina\ con\ a\}.$

Per semplificare la scrittura delle espressioni regolari possiamo sfruttare:
- Eliminazione del simbolo della concatenazione, scrivendo (st) anziché $(s \cdot t)$
- Uso di precedenza tra operatori: $*>\cdot>+$
- Eliminazione parentesi inutili

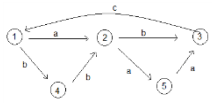
Es. $\big(a+(b\cdot(c\cdot d)^*)\big) = a + b(cd)^*$ che tradotto in linguaggio diventa $L(a) \cup L(b) \circ \big(L(c) \circ L(d)\big)^*$ ovvero il linguaggio è costituito o da una stringa a o da una stringa che inizia con b e prosegue con una ripetizione della stringa cd (può anche non essere presente dato che l'operazione * include anche la stringa vuota. Ad esempio abcdcdcd appartiene al linguaggio.

Le espressioni regolari possono descrivere anche speciali grafi.
Es.



Possiamo rappresentare il grafo di tutti i percorsi possibili per arrivare a 2 con:
(a+bb)(cdb)* con (cdb)* che può essere percorso più di una volta

Es.



Possiamo rappresentare il grafo di tutti i percorsi possibili per tornare e a 1 con:
$\big((a+bb)(b+aa)c\big)^*$

Tali linguaggi sono chiamati linguaggi regolari e si possono caratterizzare anche mediante grammatiche e macchine molto semplici.

Grammatiche di Chomsky
Da un punto di vista matematico, un linguaggio è un insieme di stringhe su un dato alfabeto, le quali stringhe in ambito informatica devono però essere caratterizzate da determinate proprietà, come quella del poter essere analizzata da un compilatore. Con il termine *grammatica* si intende un formalismo che permette di definire un insieme di stringhe mediante l'imposizione di un particolare metodo per la loro costruzione. Tale costruzione si effettua partendo da una stringa particolare e *riscrivendo* via via parti di essa secondo una qualche regola specificata nella grammatica stessa.
Le *grammatiche di Chomsky* furono introdotte dal linguista Noam Chomsky con lo scopo di rappresentare i procedimenti sintattici elementari che sono alla base della costruzione di frasi della lingua inglese. Pur essendosi presto rivelate uno strumento inadeguato per lo studio del linguaggio naturale, le grammatiche formali hanno un ruolo fondamentale nello studio delle proprietà sintattiche dei programmi e dei linguaggi di programmazione.
Una grammatica formale G è una *quadrupla* $G =< V_T, V_N, P, S >$ in cui:
1. V_T è un insieme finito e non vuoto di caratteri terminali (a,b,c);
2. V_N è un insieme finito e non vuoto di caratteri non terminali (A,B,C);
3. P è una relazione binaria di cardinalità finita su $(V_T \cup V_N)^* \circ V_N \circ (V_T \cup V_N)^* \times (V_T \cup V_N)^*$detta *insieme delle regole di produzione*.
4. S è detto assioma ed è il *simbolo non terminale di inizio*, ossia la categoria sintattica più generale.

Le produzioni di una grammatica rappresentano le regole mediante le quali una stringa non composta tutta di caratteri terminali può venire trasformata (riscritta) in un'altra.

Sia $V = V_T \cup V_N$, una regola di produzione $\alpha \to \beta$, in cui α appartiene a $V^* \circ V_N \circ V^*$, e β appartiene a V^*, significa che la sottostringa α può essere rimpiazzata con la sottostringa β.
Es. $aC \to baB$

Es. Data la grammatica $G =< \{a,b,c\}, \{S,B,C,F,C\}, P, S >$ le regole di produzione P sono:
1. $S \to aSBC$
2. $CB \to BC$
3. $SB \to bF$
4. $FB \to bF$
5. $FC \to cG$
6. $GC \to CG$
7. $G \to \varepsilon$

da un carattere non terminale (ad esempio S) possiamo ricondurci ai caratteri terminali della grammatica semplicemente sostituendo:
$$\boldsymbol{S} \to a\boldsymbol{SB}C \to ab\boldsymbol{FC} \to abc\boldsymbol{G} \to abc$$
Questa grammatica genera il linguaggio $\{a^n b^n c^n | n \geq 1\}$.
I simboli terminali che abbiamo trovato dalle regole di produzione possono inoltre essere scritte nella formula $a^n b^n$, ovvero tutte le stringhe formate da n ripetizioni di a seguite da n ripetizioni di b, ad esempio $aaabbb = a^3 b^3$.

L' $\boldsymbol{\varepsilon}$ **−regola** è una regola del tipo $\alpha \to \varepsilon$, do ve $\alpha \in V^* \circ V_N \circ V^*$;
Data una grammatica $G =< V_T, V_N, P, S >$ la **derivazione diretta** è una relazione su $(V^* \circ V_N \circ V^*) \times V^*$, rappresentata dal simbolo $\Longrightarrow_G$.

Il linguaggio generato da una grammatica G è l'insieme $L(G) = \{x | x \in V_T^* \wedge S \Longrightarrow_G^* x\}$
Il linguaggio generato da una grammatica formale è dunque un insieme di stringhe di caratteri terminali, ognuna delle quali si può ottenere a partire dall'assioma mediante l'applicazione di un numero finito di passi di derivazione diretta.

Due grammatiche G_1 e G_2 si dicono **equivalenti** se esse generano lo stesso linguaggio $L(G_1) = L(G_2)$.

Le grammatiche *Chomsky* possono generare diversi classi di linguaggi. Esse sono basate su restrizioni sul tipo di produzione:

Grammatiche di tipo 0

Le grammatiche di tipo 0, dette anche *non limitate*, descrivono produzioni del tipo:

$$\alpha \to \beta, \alpha \in V^* \circ V_N \circ V^*, \beta \in V^*$$

(una stringa alfa con caratteri terminali o non terminali concatenati ad almeno uno non terminale concatenati ad altri caratteri terminali o non terminali, diventa una stinga beta con caratteri terminali).

Es. $G =< \{a,b\}, \{S,A\}, P, S >$, in cui P è

$S \to aAb$

$aA \to aaAb$

$A \to \varepsilon$

È una grammatica di tipo 0 e genera il linguaggio $L = \{a^n b^n | n \geq 1\}$.

Dato che le grammatiche di tipo 0 sono le più generali, tutti i linguaggi generati da una grammatica di *Chomsky* sono di tipo 0.

Grammatiche di tipo 1

Le grammatiche di tipo 1, dette anche *contestuali*, descrivono produzioni del tipo:

$$\alpha \to \gamma, \alpha \in V^* \circ V_N \circ V^*, \gamma \in V^+ \; con \; |\alpha| \leq |\gamma|$$

Es. $S \to aSa|aAb|aAa$

$aA \to aa$

$Ab \to aab$

È una produzione di una grammatica di tipo 1.

Le grammatiche di tipo 1 generano linguaggi contestuali.

Grammatiche di tipo 2

Le grammatiche di tipo 2, dette anche *non contestuali*, descrivono produzioni del tipo:

$$A \to \beta, A \in V_N, \beta \in V^+$$

cioè produzioni in cui ogni non terminale A può essere riscritto in una stringa β indipendentemente dal contesto in cui esso si trova.

Es. $S \to aSb \mid ab$

Le grammatiche di tipo 2 generano linguaggi non contestuali.

Grammatiche di tipo 3

Le grammatiche di tipo 3, dette anche *lineari destre* o *regolari*, descrivono produzioni del tipo:

$$A \to \delta, A \in V_N, \delta \in (V_T \circ V_N) \cup V_T$$

Es. $G =< \{a,b\}, \{S\}, P, S >$ ha le seguenti produzioni:

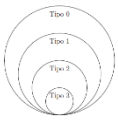
$S \to aS$

$S \to b$

È una grammatica di tipo 3 e genera il linguaggio regolare $L = \{a^n b | n \geq 0\}$.

Le grammatiche di tipo 3 generano linguaggi regolari.

Gerarchia di Chomsky



Si verifica immediatamente che per ogni $0 \leq n \leq 2$, ogni grammatica di tipo n + 1 è anche di tipo n, e pertanto l'insieme dei linguaggi di tipo n contiene tutti i linguaggi di tipo n+1, formando quindi una gerarchia, detta *Gerarchia di Chomsky*.

Un linguaggio L viene detto strettamente di tipo n se esiste una grammatica G di tipo n che genera L e non esiste alcuna grammatica G' di tipo m>m che possa generarlo.

Forma normale di Backus

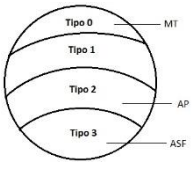
Per definire i linguaggi di programmazione vengono adottate grammatiche non contestuali rappresentati tramite una notazione specifica denominate *Forma Normale di Backus* (*Backus Normal Form, BNF*).

La BNF è una notazione per grammatiche context free resa più espressiva mediante i seguenti accorgimenti:

1. I simboli non terminali sono costruiti da stringhe racchiuse tra parentesi acute <...>
2. Il segno di produzione → viene sostituito dal simbolo ::=
3. Le parentesi graffe {...} vengono impiegate per indicare l'iterazione illimitata (0,1,2,...,n). E inoltre con $\{\dots\}^n$ indichiamo l'iterazione per un numero di volte pari a n.
   Es. $A ::= bA|a$ diventa $A ::= \{b\}a$
   $A ::= x|xy|xyy|xyyy|xyyyy$ diventa $A ::= x\{y\}^4$
4. Le parentesi quadre [..] vengono utilizzate per indicare l'opzionalità
   Es. $A ::= xy|y$ diventa $A ::= [x]y$
5. Le parentesi tonde vengono utilizzate per indicare la fattorizzazione
   Es. $A ::= xu|xv|xy$ diventa $A ::= x(u|v|y)$

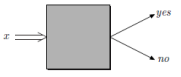
Accettazione e riconoscimento dei linguaggi

Il problema di stabilire se un programma scritto in un linguaggio di programmazione sia sintatticamente corretto può essere formalizzato come il problema di decidere se una particolare stringa (appunto, il programma sorgente) appartenga o meno a un determinato linguaggio. Per tutti i vari tipi di linguaggi esistenti vengono adoperate delle diverse macchine astratte specifiche chiamate automi che si occupano della computazione delle stringhe del linguaggio interessato:



Automa

Dispositivo astratto che, data una stringa x fornitagli in input, può eseguire una *computazione*, ed eventualmente se la computazione termina, restituisce un valore booleano (vero o falso). Essi sono utilizzati in informatica poiché tali macchine (astratte) sono programmate con dei logaritmi che permettono di riconoscere un linguaggio. Un automa è costituito da:



- Macchina: ad ogni instante assume uno *stato (q)* che rappresenta l'informazione associata al funzionamento interno.
- Nastro: sequenza di celle sulla quale l'automa memorizza le informazioni sotto forma di stringhe di caratteri.
- Testina: si muove lungo il nastro e legge i caratteri inseritovi sopra.

Il funzionamento di un automa è definito rispetto a due concetti:

Configurazione: l'insieme delle informazioni che determinano il comportamento futuro dell'automa. Essa è composta dalle seguenti informazioni:

- Stato interno dell'automa;
- Contenuto di tutti i nastri di memoria;
- Posizione di tutte le testine sui nastri.

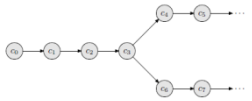
Transizione: induce una *relazione di transizione* tra configurazioni che associa ad una configurazione un'altra *configurazione successiva*.
Per ogni automa *A*, date due configurazioni c_i e c_j di A useremo la notazione $c_i \vdash_A c_j$ per indicare che c_i e c_j sono correlate dalla relazione di transizione che indica che da c_j deriva c_i per effetto dell'applicazione della funzione di transizione di *A*.
Possiamo vedere una computazione eseguita da un automa *A* a partire da una configurazione iniziale c_0 come una sequenza di configurazioni $c_0, c_1, c_2, \dots : \forall i = \{0,1,2,\dots\}\ c_i \vdash_A c_{i+1}$. Indicheremo con la notazione $\vdash_A^*$ la chiusura transitiva e riflessiva della relazione $\vdash_A$.
Se la lunghezza di configurazioni $c_0, c_1, c_2, \dots, c_n$ ha lunghezza finita, e non esiste nessun'altra configurazione $c_n \vdash_A c$, allora diciamo che la computazione *termina*.

Computare
Computare significa ridurre un'informazione da una forma implicita ad una forma esplicita in modo effettivo, ovvero rielaborare un insieme di dati al fine di presentarli in una forma di più diretta comprensione.
Es. 3 + 2 è una forma implicita per rappresentare il 5, mentre 5 è una forma esplicita per rappresentare il 5.

Automa deterministico
Un automa è detto deterministico se ad ogni stringa di input associa una sola stringa di configurazioni. Un automa deterministico quindi può eseguire una sola computazione: se tale computazione termina in una configurazione di accettazione, allora la stringa viene accettata.

Dato un automa deterministico *A* e data una stringa x di input ad *A*, se indichiamo con $c_0(x)$ la configurazione iniziale di A corrispondente alla stringa x, avremo che A accetta x se e solo se esiste una configurazione di accettazione c di A per la quale $c_0(x) \vdash_A^* c$. Il linguaggio accettato da A sarà allora l'insieme $L(A)$ di tutte le stringhe x accettate da A. Se inoltre ogni stringa viene o accettata o rifiutata, allora diciamo che il linguaggio $L(A)$ è riconosciuto da A.

Automa non deterministico
Un automa è detto non deterministico se esso associa ad ogni stringa di input un numero qualunque di computazioni.



In questo caso la funzione di transizione associa a qualche configurazione *c* più di una configurazione successiva. Con il termine *grado di non determinismo* di un automa indichiamo il massimo numero di configurazioni che la funzione di transizione associa ad una configurazione. L'automa deterministico è dunque formato da un *albero di computazione*, con i nodi che rappresentano le configurazioni dell'automa; la astringa viene accetta se *una qualunque* delle computazioni definite è di accettazione, mentre non lo viene se *tutte* le possibili computazioni che terminano non sono di accettazione.

Automi e classi di linguaggi
Un problema decisionale (o un linguaggio) è detto *decidibile* se esiste un algoritmo (o un automa) che per ogni istanza del problema risponde correttamente VERO oppure FALSO. In caso contrario il problema è detto indecidibile.
Un problema decisionale (o un linguaggio) è detto *semidecidibile* se esiste un algoritmo (o un automa) che per tutte e sole le istanze positive del problema risponde correttamente VERO.
Esistono vari tipi di automi che possono essere utilizzati per risolvere problemi di decisione relativi alle diverse classi di linguaggi:
- Linguaggi di tipo 3: esistono dispositivi di riconoscimento che operano con memoria costante e tempo lineare (*automi a stati finiti*).
- Linguaggi di tipo 2: il riconoscimento avviene sempre in tempo lineare ma con dispositivi di tipo non deterministico dotati di una memoria a pila (*automi a pila non deterministici*).
- Linguaggi di tipo 1: dato che l'esigenza di tempo e memoria è ancora maggiore, per essi il riconoscimento può essere effettuato con una macchina non deterministica che fa uso di una quantità di memoria che cresce linearmente con la lunghezza della stringa da esaminare (*automa lineare*).
- Linguaggi di tipo 0: si farà riferimento a un dispositivo di calcolo costituito da un automa che opera con quantità di tempo e di memoria potenzialmente illimitate (*macchina di Turing*).

Automi a stati finiti (ASF)
Questi automi sono dispositivi di riconoscimento per linguaggi di tipo 3, e si possono pensare come macchine che leggono la stringa di input e la elaborano facendo uso di un elementare meccanismo di calcolo e di una memoria finita e limitata. L'esame della stringa avviene un carattere alla volta mediante delle configurazioni che comportano lo spostamento delle destina sul carattere successivo e l'aggiornamento della memoria.

Automi a stati finiti deterministici (ASFD)
Un automa a stati finiti deterministico è una quintupla $A =< \Sigma, Q, \delta, q_0, F >$ dove
- $\Sigma = \{a_1, \dots, a_n\}$ è l'*alfabeto di input*;
- $Q = \{q_0, \dots, q_m\}$ è un *insieme finito e non vuoto di stati*;
- $F \subseteq Q$ è un *insieme di stati finali*;
- $q_0 \in Q$ è lo *stato iniziale*;
- $\delta: Q \times \Sigma \to Q$ è la *funzione di transizione* che prende una coppia <stato, carattere in input> e restituisce uno stato successivo.

È possibile definire anche *la funzione di transizione estesa alle stringhe* con $\bar{\delta}: Q \times \Sigma^* \to P(Q)$, quando
- $\bar{\delta}(q, \varepsilon) = q$ la transizione da uno stato che prende in input un carattere vuoto da lo stato stesso
- $\bar{\delta}(q, ax) = \bar{\delta}(\delta(q, a), x)$ con $x \in \Sigma^*$ e $a \in \Sigma$. La transizione da uno stato che prende in input due caratteri diversi da lo stesso stato in cui si perviene leggendo lo stato prendendo a in input e dalla stringa x

Un ASFD esegue una computazione nel modo seguente: a partire da q_0, l'automa ad ogni passo legge il carattere successivo della stringa di input ed applica la funzione δ per determinare lo stato successivo; terminata la lettura della stringa, essa viene accettata se lo stato attuale è uno stato finale, altrimenti viene rifiutata.

Dato un automa a stati finiti $A =< \Sigma, Q, \delta, q_0, F >$, una configurazione di A è una coppia (a, x) con $q \in Q$ e $x \in \Sigma^*$.
Una configurazione $< q, x >, q \in Q$ ed $x \in \Sigma^*$, di A è detta:
- Iniziale se $q = q_0$;
- Finale se $x = \varepsilon$;
- Accettante se $x = \varepsilon$ e $q \in F$.

Dunque dato un ASFD $A =< \Sigma, Q, \delta, q_0, F >$, una stringa $x \in \Sigma^*$ è accettata da A se e solo $(q_0, x) \vdash_A^* (q, \varepsilon)$, con $q \in F$, e possiamo definire il linguaggio riconosciuto da A come: $L(A) = \{x \in \Sigma^* | (q_0, x) \vdash_A^* (q, \varepsilon), q \in F\}$.
Il linguaggio riconosciuto da un ASFD A è l'insieme $L(A) = \{x \in \Sigma^* | \bar{\delta}(q_0, x) \in F\}$.

Rappresentazione di un ASF

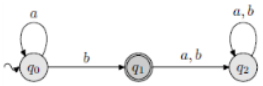
Dato il linguaggio $\{a^n b | n \geq 0\}$ generato da $S \rightarrow aS|b$, l'automa che lo riconosce è l'automa $< \{a,b\}, \{q_0, q_1, q_2\}, \delta, q_0, \{q_1\} >$ dove:

- Definiamo Il comportamento dell'automa rappresentandolo mediante la *tabella di transizione* alle cui righe vengono associati gli stati, mentre alle colonne i caratteri di input, ed i cui elementi rappresentano il risultato dell'applicazione della δ allo stato identificato dalla riga ed al carattere associato alla colonna della tabella.

δ	a	b
q_0	q_0	q_1
q_1	q_2	q_2
q_2	q_2	q_2

Se sono allo stato q_0 e leggo il carattere b, mi ritroverò nello stato q_1, ecc.

- Un'altra rappresentazione è costituita dal così detto *diagramma degli stati*, in cui l'automa è rappresentato mediante un grafico dove i nodi rappresentano gli stati mentre gli archi sono le transizioni; gli stati finali sono rappresentati da nodi con un doppio circolo, mentre quello iniziale è individuato tramite una freccia entrante.



L'automa rimane nello stato q_0 fin quanto legge a, quanto legge b si porta in q_1 che è uno stato finale, se però la stringa non è terminata si arriva allo stato q_2 dove però avremo uno stato di errore visto che non è finale.

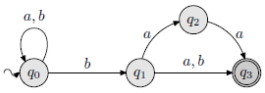
Automi a stati finiti non deterministici (ASFND)

Sono automi che possono effettuare una transizione da uno stato a più stati; essi sono utili per studiare le proprietà dei linguaggi regolari.
Un automa a stati finiti non deterministico è una quintupla $A_N =< \Sigma, Q, \delta_N, q_0, F >$ dove

- $\Sigma = \{a_1, \dots, a_n\}$ è l'*alfabeto di input*;
- $Q = \{q_0, \dots, q_m\}$ è un *insieme finito e non vuoto degli stati interni*;
- $F \subseteq Q$ è un *insieme di stati finali*;
- $q_0 \in Q$ è lo *stato iniziale*;
- $\delta_N: Q \times \Sigma \rightarrow P(Q)$ è una *funzione (parziale) di transizione* che ad ogni coppia <stato, carattere in input> determina l'insieme degli stati successivi.

δ	a	b
q_0	$\{q_0\}$	$\{q_0, q_1\}$
q_1	$\{q_2, q_3\}$	$\{q_3\}$
q_2	$\{q_3\}$	$\emptyset$
q_3	$\emptyset$	$\emptyset$

Come si può notare dalla tabella di transizione le coppie (q_0, b) e (q_1, a) forniscono più stati, corrispondenti a diverse possibili continuazioni di una computazione.
Un ASFND definisce quindi, data una stringa di input, un insieme di computazioni.
Si può definire anche qui la *funzione di transizione estesa alle stringhe* come



$\bar{\delta}: Q \times \Sigma^* \rightarrow P(Q)$ solo che $\bar{\delta}_N(q, ax) = \cup \bar{\delta}_N(p, x)$ con $p \in \delta_N(q, a)$.

Es. $\delta_N(q_1, a) = \{q_1, q_2\}$, $\delta_N(q_1, b) = \{q_2, q_3\}$, $\delta_N(q_2, b) = \{q_4, q_5\}$ darà $\bar{\delta}_N(q_1, ab) = \{q_2, q_3\} \cup \{q_4, q_5\}$

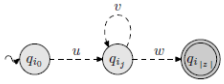
Una stringa viene accetta da un AFND se almeno una delle computazioni definite per la stringa stessa è di accettazione: una stringa x è accettata se $(\{q_0\}, x) \vdash^* (Q, \varepsilon)$, dove $Q \subseteq Q$ e $Q \cap F \neq \emptyset$.
Possiamo allora definire il linguaggio $L(A)$ accettato da un ASFND A nel modo seguente: $L(A) = \{x \in \Sigma^* | (\{q_0\}, x) \vdash^* (Q, \varepsilon), Q \cap F \neq \emptyset\}$.
Il linguaggio riconosciuto da un ASFND A_N è l'insieme: $L(A_N) = \{x \in \Sigma^* | \bar{\delta}_N(q_0, x) \cap F \neq \emptyset\}$.

Pumping lemma

Per ogni linguaggio regolare *L* esiste una costante *n* tale che se $z \in L$ e $|z| \geq n$ allora possiamo scrivere $z = uvw$, con $|uv| \leq n, |v| \geq 1$ e ottenere che $uv^i w \in L \; \forall i \geq 0$.
Dato un linguaggio regolare *L*, sia l'ASFD che lo riconosce $A =< \Sigma, Q, \delta, q_0, F >$ avente il minimo numero possibile di stati $n = |Q|$. Sia *z* una stringa appartenente ad *L*, con $|z| \geq n$. Supponiamo che $q_{i0}, q_{i1}, \dots, q_{i|z|}$ sia la sequenza di stati attraversata da *A* durante il riconoscimento di *z*, con $q_{i0} = q_0 e \; q_{i|z|} \in F$. Dal momento che $|z| \geq n$ deve esistere almeno uno stato cui l'automa porta almeno due volte durante la lettura di *z*:

- Sia *j* il minimo valore tale che q_{ij}, viene attraversato almeno due volte;
- Sia *u* il più breve prefisso di *z* tale che $\bar{\delta}(q_0, u) = q_{ij}$
- Sia $z = ux$
- Sia *v* il più breve prefisso di *x* tale che $\bar{\delta}(q_{ij}, v) = q_{ij}$
- Sia $x = vw$

Per ogni $i \geq 0$ abbiamo allora:
$\bar{\delta}(q_0, uv^i w) =$
$\bar{\delta}(\bar{\delta}(q_0, u), v^i w)$
$= \bar{\delta}(q_{ij}, v^i w)$
$= \bar{\delta}(\bar{\delta}(q_{ij}, v), v^{i-1} w)$
$= \bar{\delta}(q_{ij}, v^{i-1} w)$
$= \dots$
$= \bar{\delta}(q_{ij}, w)$
$= q_{i|z|}$



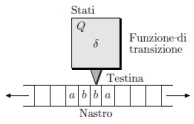
Il che mostra che ogni stringa del tipo $uv^i w \in L$.
Il pumping lemma evidenzia uno dei principali limiti degli automi finiti: essi non possiedono la capacità di contare. Infatti essi possono memorizzare mediante i propri stati solo un numero finito di diverse situazioni, mentre non possono memorizzare numeri.

Modelli computazionali e teoria della calcolabilità

Le *Macchine di Turing* (MT) sono il modello di calcolo di riferimento fondamentale nell'ambito della teoria della calcolabilità e della teoria della complessità computazionale.
Esse furono introdotte dal logico inglese Alan Turing con l'obbiettivo di formalizzare il concetto di calcolo allo scopo di stabilire l'esistenza di metodi algoritmici per il riconoscimento dei teoremi nell'ambito dell'aritmetica. Il modello definito da Turing ha assunto un ruolo molto importante per lo studio dei fondamenti teorici dell'informatica perché in tale modello si associa una elevata semplicità di struttura e di funzionamento ad un potere computazionale che è, fino ad oggi, ritenuto il massimo potere computazionale realizzabile da un dispositivo di calcolo automatico.
Così come gli automi a stati finiti e quelli a pila, la macchina di Turing è definita come un dispositivo operante su stringhe contenute su una memoria esterna (nastro) mediante passi elementari, definiti da una opportuna funzione di transizione.

Macchine di Turing a nastro singolo

Nella sua versione più tradizionale una macchina di Turing si presenta come un dispositivo che accede ad un nastro potenzialmente illimitato diviso in celle contenenti ciascuna un simbolo appartenente ad un dato alfabeto Γ, ampliato con il *carattere speciale* ~~b~~ (blank) che rappresenta la situazione di cella non contenente caratteri. La macchina di Turing opera su tale nastro tramite una testina, la quale può scorrere su di esso in entrambe le direzioni. Su ogni cella la testina può leggere o scrivere caratteri appartenenti all'alfabeto Γ oppure il simbolo ~~b~~.



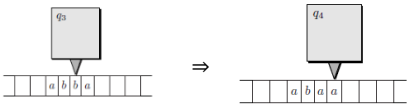
Ad ogni instante la macchina si trova in uno *stato* appartenente ad un insieme finito *Q*, ed il meccanismo che fa evolvere la computazione della macchina è una funzione di transizione δ la quale porta la macchina da uno stato all'altro e determinando la scrittura del carattere su una cella e lo spostamento della testina stessa.

Una macchina di Turing deterministica (MTD) è una sestupla $M =< \Gamma, \text{~~b~~}, Q, q_0, F, \delta >$ dove:

- Γ è *l'alfabeto dei simboli di nastro*;
- ~~b~~ $\in \Gamma$ è un *carattere speciale* denominato blank;
- Q è un *insieme finito e non vuoto* di stati;
- $q_0 \in Q$ è lo *stato iniziale*;
- $F \subseteq Q$ è l'*insieme degli stati finali*;
- δ è la *funzione (parziale) di transizione* definita come: $\delta: (Q - F) \times (\Gamma \cup \{\text{~~b~~}\}) \times \{d, s, i\}$

  in cui *d* = spostamento a destra della tesina, *s* = spostamento a sinistra della tesina, *i* = immobile.

Es.

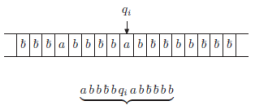


Transizione definita da $\delta(q_3, b) = (q_4, a, d)$.

Dato un alfabeto Γ tale che b ∉ Γ, si indicherà con $\overline{\Gamma}$ l'insieme $\Gamma \cup \{\text{~~b~~}\}$.

Le MTD possono venire utilizzate sia come strumenti per il calcolo di funzioni, sia per stabilire se esse appartengano ad un certo linguaggio oppure no. Le macchine usate per accettare stringhe vengono dette di tipo *riconoscitore*, mentre quelle usate per calcolare funzioni vengono dette di tipo *trasduttore*.

Configurazioni e transizioni delle Macchine di Turing



La configurazione di una macchina di Turing è l'insieme delle informazioni costituito dal contenuto del nastro, dalla posizione della testina e dallo stato corrente. Anche se la macchina di Turing è definita come un dispositivo operante su di un nastro infinito, in ogni momento solo una porzione finita di tale nastro contiene simboli diversi da ~~b~~. Per tale ragione, si conviene di rappresentare nella configurazione la più piccola porzione (finita) di nastro contenente tutti i simboli non ~~b~~, includendo obbligatoriamente la cella su cui è posizionata la testina.

Si definisce *configurazione di una macchina di Turing* (o configurazione istantanea) con alfabeto di nastro Γ ed insieme degli stati Q, una stringa $c = xqy$ con:

1. $x \in \Gamma\overline{\Gamma}^* \cup \{\varepsilon\}$;
2. $q \in Q$;
3. $y \in \overline{\Gamma}^*\Gamma \cup \{\text{~~b~~}\}$.

La stringa xqy vuol dire che lo stato attuale è q e la testina è posizionata sul carattere y; se $x = \varepsilon$ abbiamo che a sinistra della testina compaiono solo simboli ~~b~~, mentre per y = ~~b~~ sulla cella attuale e a destra della testina compaiono soltanto simboli ~~b~~.

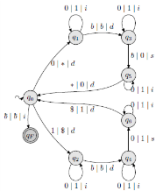
Indicheremo con L_Γ il linguaggio $\Gamma\overline{\Gamma}^* \cup \{\varepsilon\}$ delle stringhe che possono comparire a sinistra del simbolo di stato in una configurazione, e con R_Γ il linguaggio $\overline{\Gamma}^*\Gamma \cup \{\text{~~b~~}\}$ delle stringhe che possono comparire alla sua destra.

Un particolare tipo di configurazione è la *configurazione iniziale* (q_0), la quale è una configurazione che data una stringa $x \in \Gamma^*$, rappresenti stato e posizione della testina all'inizio di una computazione su input x. Una configurazione $c = xqy$ si dice iniziale se $x = \varepsilon, q = q_0, y \in \Gamma^* \cup \{\text{~~b~~}\}$.

Altro tipo di configurazione di interesse è quella *finale*. Una configurazione $c = xqy$ con $x \in L_\Gamma, y \in R_\Gamma$ si dice finale se $q \in F$. Quindi, una macchina di Turing si trova in una configurazione finale se il suo stato attuale è uno stato finale, indipendentemente dal contenuto del nastro e dalla posizione della testina.

Sappiamo che una funzione di transizione può essere rappresentata mediante matrici di transizione e grafi di transizione. Nel caso della macchina di Turing, le colonne della matrice di transizione corrispondono ai caratteri osservabili sotto la testina (elementi di $\overline{\Gamma}$) e le righe ai possibili stati interni della macchina (elementi di Q), mentre gli elementi della matrice (nel caso di macchina di Turing deterministica) sono triple che rappresentano il nuovo stato, il carattere che viene scritto e lo spostamento della testina. Nella rappresentazione della funzione di transizione mediante grafo i nodi sono etichettati con gli stati e gli archi con una tripla composta dal carattere letto, il carattere scritto e lo spostamento della testina.

	0	1	*	$	b
q_0	$(q_1, *, d)$	$(q_2, \$, d)$	-	-	(q_F, b, i)
q_1	$(q_1, 0, d)$	$(q_1, 1, d)$	-	-	(q_3, b, d)
q_2	$(q_2, 0, d)$	$(q_2, 1, d)$	-	-	(q_4, b, d)
q_3	$(q_3, 0, d)$	$(q_3, 1, d)$	-	-	$(q_5, 0, s)$
q_4	$(q_4, 0, d)$	$(q_4, 1, d)$	-	-	$(q_6, 1, s)$
q_5	$(q_5, 0, s)$	$(q_5, 1, s)$	$(q_0, 0, d)$	-	(q_5, b, s)
q_6	$(q_6, 0, s)$	$(q_6, 1, s)$	-	$(q_0, 1, d)$	(q_6, b, s)
q_F	-	-	-	-	-



Es. La regola $\delta(q_1, a) = (q_2, b, s)$ applicata alla configurazione abb~~b~~b$q_1$ab~~b~~b porta alla configurazione abb~~b~~b$q_2$bbbbbb

Computazioni di macchine di Turing

Le MTD sono delle macchine adeguate per affrontare il problema del riconoscimento. Dato un *alfabeto di input* $\Sigma \subseteq \Gamma$, una macchina di Turing può essere vista come un dispositivo che classifica le stringhe in Σ^* in funzione del tipo di computazione indotto.

Se la funzione di transizione fa passare dalla configurazione c_i alla configurazione c_j scriviamo $c_i \vdash c_j$. Una **computazione** è una sequenza eventualmente infinita di configurazioni $< c_1, c_2, \dots, c_i, c_{i+1}, \dots >$ tali che: $c_1 \vdash c_2 \vdash \cdots \vdash c_i \vdash c_{i+1} \vdash \cdots$. Una computazione finita $c_1 \vdash c_2 \vdash \cdots \vdash c_n$ è **massimale** se non esiste una configurazione c tale che $c_n \vdash c$; in tal caso c_n è una configurazione finale o è una configurazione in cui la funzione di transizione non è definita, e quindi il calcolo non può proseguire.

Riconoscimento e accettazioni di linguaggi

A differenza degli ASF (automi a stati finiti) e AP (automi a pila) che riconoscono rispettivamente linguaggi di tipo 3 e 2, le MT non sono sempre in gradi di riconoscere un linguaggio di tipo 0, ma in alcuni casi possono solo accettarlo.

- Una macchina di Turing $\mathcal{M}$ **riconosce** un linguaggio L se per ogni $x \in \Sigma^*$, $\mathcal{M}$ è in grado di stabilire se $x \in L$ o no.
- Una macchina di Turing $\mathcal{M}$ **accetta** un linguaggio L se per tutte e sole le $x \in L$, $\mathcal{M}$ è in grado di stabilire tale appartenenza, ma se $x \notin L$, $\mathcal{M}$ non garantisce un comportamento prestabilito.

Una computazione massimale è **rifiutante** se c_0 è iniziale e c_n non è finale, una computazione infinita non corrisponde a nessun responso.

Sia $M =< \Gamma, \text{~~b~~}, Q, q_0, F, \delta >$ una MT, diciamo che M **riconosce** (decide) un linguaggio $L \in \Sigma^*$ se e solo se per ogni $x \in \Sigma^*$ esiste una computazione massimale $q_0 x \vdash^* \alpha q \beta$, con $\alpha \in (\Sigma_{\text{~~b~~}})^*$, e $\beta \in (\Sigma_{\text{~~b~~}})^+$, e dove $q \in F$ se e solo se $x \in L$. Un linguaggio riconosciuto è detto **decidibile**.

Sia $M =< \Gamma, \text{~~b~~}, Q, q_0, F, \delta >$ una MT, diciamo che M **accetta** un linguaggio $L \in \Sigma^*$ se e solo se $L = \{x \in \Sigma^* | q_0 x \vdash^* \alpha q \beta\}$, con $\alpha \in (\Sigma_{\text{~~b~~}})^*$, e $\beta \in (\Sigma_{\text{~~b~~}})^+$ e $q \in F$. Un linguaggio accettato è detto **semidecidibile**.

Calcolabilità secondo Turing

Un linguaggio è detto decidibile secondo Turing (T-decidibile) se esiste una macchina di Turing che lo riconosce.
Un linguaggio è detto semidecidibile secondo Turing (T-semidecidibile) se esiste una macchina di Turing che lo accetta.
Ogni linguaggio T-decidibile è anche T-semidecidibile, ma non il viciversa.
Una funzione è detta calcolabile secondo Turing (T-calcolabile) se esiste una macchina di Turing che la calcola.

Calcolo di funzioni e macchine di Turing deterministiche

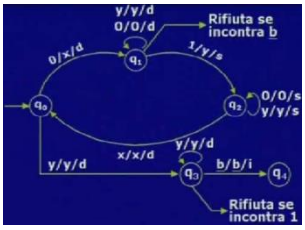
Consideriamo le macchine di Turing come dei *trasduttori*, cioè dispositivi generali capaci di realizzare il calcolo di funzioni parziali definite su domini qualunque. Dato un trasduttore $M =< \Gamma, \text{b}, Q, q_0, F, \delta >$ ed una funzione $f: \Sigma^* \to \Sigma^* (\Sigma \subseteq \Gamma)$, M calcola la funzione f se e solo $\forall x \in \Gamma^*$:

1. Se $x \in \Sigma^*$ e $f(x) = y$ allora $q_0 x \vdash_M^* x\text{b}qy, con\ q \in F$;
2. Se $x \notin \Sigma^*$

Funzionamento di una MT

Es. MT che riconosce $0^n 1^n\ (n \geq 1)$ $M =< \{0,1\}, \overline{b}, Q, q_0, \{q_4\}, \delta >$ configurazione iniziale: $q_0 0011$

La MT marca via via con un X gli 0 e con un Y i corrispondenti 1 e accetta se gli 0 e gli 1 sono in numero uguale, rifiuta se sono in numero diverso



$q_0 0011$ la macchina legge il carattere 0, stampa x e si sposta a destra
$x q_1 011$ la macchina legge 0, rimane su q_1 ma si sposta a destra
$x0q_1 11$ la macchina legge 1, stampa y e si sposta a sinistra
$x q_2 01$ la macchina si trova su q_2, legge 0 e continua a spostare la testina a destra
$q_2 x0y1$ la macchina individua un x, lo sposta a destra e ritorna allo stato q_0
$x q_0 0y1$ adesso la macchina ricomincerà il suo ciclo scavalcando le y e marcando l'uno con y
$xxq_1 y1, xxyq_1 1, xxq_2 yy$. Nello stato q_3 la testina riconoscerà di aver marcato tutte i 0 e gli 1, trova un simbolo blanck e va nello stato q_4

Una MT opera nel seguente modo: partendo dall'assioma S la macchina applica via via, in modo non deterministico, tutte le produzioni applicabili, e dopo aver applicato la produzione M verifica se la forma di frase ottenuta coincide con la stringa da generare x. In tal caso M termina in uno stato finale.

Supponiamo di avere una MT che accetta un linguaggio L. Vediamo come si può costruire una grammatica G di tipo 0 per L:
Se la macchina accetta L vuol dire che, per tutte e sole le $x \in L$, essa realizza la computazione $q_0 x \vdash u q_f w$ (da uno stato iniziale a finale); La grammatica G simula le computazione di M e genera una stringa x se e soltanto se essa viene accettata.
L'alfabeto non terminale di G è costituito da simboli ausiliari e da simboli che rappresentano il contenuto di due piste entrambe dotate di caratteri blanck: $\begin{bmatrix}\sigma_i \\ \sigma_j\end{bmatrix}$

La grammatica genera un qualunque stringa di x di Σ^* sulle due piste, simula M sulla posta inferiore, se la pista inferiore corrisponde ad una configurazione finale trasforma la forma di frase ottenuta nella stringa x.

Es. la grammatica genera $x \begin{bmatrix}\text{b} \\ \text{b}\end{bmatrix} q_0 \begin{bmatrix}a \\ a\end{bmatrix} \begin{bmatrix}b \\ b\end{bmatrix} \begin{bmatrix}a \\ a\end{bmatrix} \begin{bmatrix}\text{b} \\ \text{b}\end{bmatrix}$H la simulazione avviene sulla pista inferiore

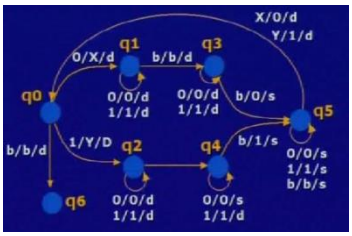
MT che calcolano funzioni

Una Mt calcola la funzione f se per tutte e sole le x nel dominio di definizione f esiste $q \in F$ (stato finale) tale che:

$$q_0 x \vdash x\text{b}qf(x)$$

ovvero: dallo stato iniziale la MT legge x e quando termina ritorna la stessa x seguita da uno spazio bianco, trova lo stato finale e calcola f(x)
Una macchina di questo tipo si chiama **trasduttore**. Una funzione parziale f per la quale esiste una MT che la calcola è detta *calcolabile secondo Turing* (T-calcolabile). Naturalmente se la f non è definita per un valore di x, la macchina può non terminare.

Es. MT che calcola f(x)=x per $x \in \{0,1\}^*$. Config. iniziale: $q_0 x$ Config. finale: $x\text{b}q_F x$ StringadiEs: 01



$q_0 01$
$x q_1 1$
$x1q_1$
$x1\text{b}q_3$
$x1q_5\text{b}0$
$xq_5 1\text{b}0$
$q_5 x1\text{b}0$
$0q_0 1\text{b}0$
$0yq_2\text{b}0$...
La testina arriva a q_6 che è lo stato finale

Macchina di Turing multinastro

Un'utile variante della macchina di Turing classica è rappresentata dalla macchina di Turing *a più nastri* o *multinastro* (*MTM),* in cui si hanno più nastri contemporaneamente accessibili in scrittura e lettura che vengono consultati e aggiornati tramite più testine.
Anche se una macchina di Turing multinastro non aggiunge potere computazionale al modello delle macchine di Turing, essa consente di affrontare determinate classi di problemi in modo particolarmente agile.
Una macchina di Tuning a k nastri è costituita da: $M^k =< \Sigma, \text{b}, Z_0, Q, q_0, F, \delta^k >$ dove:
Σ è l'alfabeto, b il carattere blank, Z_0 il carattere iniziale, Q l'insieme di stati, q_0 lo stato iniziale, F l'insieme di stati finali, δ^k la funzione di transizione che opera in questo modo: $\delta: Q \times (\Sigma_\text{b})^k \to Q \times (\Sigma_\text{b})^k \times \{d, s, i\}^k$

URM (Unlimited Register Machines, Macchine a registri illimitati)

Una URM è dotata di un array infinito di registri $R_1, R_2, R_3, \ldots$ ciascuno dei quali contiene un *numero naturale* (intero non negativo) $r_1, r_2, r_3, \ldots$

R_1	R_2	R_3	R_4	R_5	R_6	R_7	...
r_1	r_2	r_3	r_4	r_5	r_6	r_7	...

Lo *stato* di una URM è il contenuto dei suoi registri $\vec{r} \in \mathbb{N}^\infty$. Questo stato può essere modificato dall'esecuzione di un URM-programma.
Un *URM-programma* è una sequenza finita $I_1, I_2, \ldots, I_s$ (s lunghezza del programma) di istruzioni di uno dei seguenti quattro tipi:

- Reset
- Incremento
- Assegnamento
- Salto condizionato

Un *configurazione istantanea* è una coppia $(k, \vec{r}) \in \mathbb{N}^+ \times \mathbb{N}^\infty$ con

- k intero positivo, detto *contatore di programma* che rappresenta l'indice della istruzione che sta per essere eseguita quando lo stato dei registri è $\vec{r}$.
- $\vec{r}$ stato

Es. (5,(1,0,0,…)) è una configurazione istantanea.

Istruzione di reset

Indicata con $Z(n)$, $n \in N^+$, l'istruzione è $[R_n \leftarrow 0]$. Semantica: $(k,\vec{r}) \rightarrow^{Z(n)} (k+1,\overrightarrow{r^1})$ con $r_i^1 = \begin{cases} r_i, i \neq n \\ 0, i = n \end{cases}$

Istruzione di incremento

Indicata con $S(n)$, $n \in N^+$, l'istruzione è $[R_n \leftarrow R_n + 1]$. Semantica: $(k,\vec{r}) \rightarrow^{S(n)} (k+1,\overrightarrow{r^1})$ con $r_i^1 = \begin{cases} r_i, i \neq n \\ r_n + 1, i = n \end{cases}$

Istruzione di assegnamento

Indicata con $T(m,n)$, $m,n \in N^+$, l'istruzione è $[R_n \leftarrow R_m]$. Semantica: $(k,\vec{r}) \rightarrow^{T(m,n)} (k+1,\overrightarrow{r^1})$ con $r_i^1 = \begin{cases} r_i, i \neq n \\ r_m, i = n \end{cases}$

Istruzione di salto condizionato

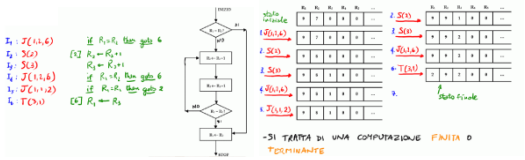
Indicata con $J(m,n,q)$, $m,n,q \in N^+$, l'istruzione è $[if\ R_m = R_n\ then\ goto\ q]$. Semantica: $(k,\vec{r}) \rightarrow^{J(m,n,q)} (k',\overrightarrow{r^1})$ con $k' = \begin{cases} q, r_m \neq r_n \\ k+1, r_m \neq r_n \end{cases}$

Computazioni

Dati

- $P = I_1, I_2, \dots, I_S$ (Programma di lunghezza s)
- $\vec{r}$ (Stato dei registri)

La computazione $P(\vec{r})$ di P a partire dallo stato iniziale $\vec{r}$ è la sequenza (finita o infinita) di configurazioni istantanee $\left(k_1,\overrightarrow{r^1}\right),\left(k_2,\overrightarrow{r^2}\right),\left(k_3,\overrightarrow{r^3}\right),\dots$ tale che $(k_1 = 1, \overrightarrow{r^1} = \vec{r})$, cioè la computazione inizia con la prima istruzione di P e con lo stato iniziale $\vec{r}$.



Notazioni utili

Sia P un programma e $\vec{r}$ uno stato (iniziale dei registri);

- Se la computazione $P(\vec{r})$ è terminale scriveremo $P(\vec{r}) \downarrow$
- Se la computazione $P(\vec{r})$ è non terminale scriveremo $P(\vec{r}) \uparrow$

Con la notazione $P(a_1, a_2, \dots, a_n)$ indicheremo la computazione $P(\vec{r})$ dove $P_1 = \begin{cases} a_i\ se\ i \leq n \\ 0\ \ altrimenti \end{cases}$

Convenzioni di input e output

Sia $P(a_1, a_2, \dots, a_n) \downarrow$. L'**output** di P su input $a_1, a_2, \dots, a_n$ è il contenuto del registro R_1 nello stato finale della computazione $P(a_1, a_2, \dots, a_n)$, e scriveremo $P(a_1, a_2, \dots, a_n) \downarrow b$ per indicare che b è l'output della computazione $P(a_1, a_2, \dots, a_n)$.

Funzioni parziali

Sia $f: A \rightarrow B$ una funzione parziale, se f è definita su a scriveremo $f(a) \downarrow$, altrimenti scriveremo $f(a) \uparrow$.
Siano $f, g: A \rightarrow B$ funzioni parziali. Scriveremo $f \cong g$ per indicare che

1) Per ogni $a \in A$: $f(a) \downarrow$ se e solo se $g(a) \downarrow$
2) Per ogni $a \in A$: se $f(a) \downarrow$ allora $f(a) = g(a)$

Sia P un programma, per ogni $n \geq 1$ poniamo $f_p^n(a_1, a_2, \dots, a_n) = \begin{cases} b\ se\ P(a_1, a_2, \dots, a_n) \downarrow b \\ \uparrow\ altrimenti \end{cases}$

La funzione parziale $f_p^n: N^n \rightarrow N$ è la funzione n-aria calcolata da P. Una funzione parziale $g: N^n \rightarrow N$ si dice URM-Calcolabile se esiste un programma URM P tale che $g \cong f_P^n$.

Linguaggi di programmazione imperativi e funzionali

Programmare significa specificare un particolare processo di computazione con un linguaggio basato su un particolare modello di computazione.

- I *linguaggi di programmazione funzionale* sono basati sul modello computazionale del **Lambda-calcolo**.
- I *linguaggi di programmazione imperativi* sono basati su modelli computazioni come le macchine di Turing o le URM.

Mentre i concetti fondamentali dei linguaggi di programmazione imperativi sono: memoria, variabile (qualcosa che è possibile da modificare), assegnamento, istruzione, iterazione; nei linguaggi funzionali questi concetti sono assenti, infatti noi abbiamo: espressione, ricorsione, evalutazione, variabile (intesa matematicamente, entità sconosciuta e astratta).

Programmazione funzionale

Un programma in linguaggio funzionale consiste in un insieme di definizioni di funzioni e espressioni. L'interprete del linguaggio valuta questa espressione come un discreto numero di passi di computazione, trasformando il suo significato in esplicito.
Noi sappiamo che una funzione è definita da un legge f, da un dominio X e un codominio Y di elementi, rappresentabile con $f: X \rightarrow Y$ con $x \in X$ e $y \in Y$, dunque $(x,y) \in f$. Due funzioni $f, g: X \rightarrow Y$ sono considerate uguali se hanno gli stessi elementi, cioè $f(x) = g(x)$ per $x \in X$.
Questo approccio è chiamato *visione estensionale delle funzioni*, poiché specifica che la sola cosa osservabile riguardo una funzione è come sono settati gli elementi.
Prima del 20° secolo le funzioni non erano definite così. Si pensava che le funzioni fossero delle *regole*, ovvero dare una funzione significava dare una regola. Due funzioni erano *estensionalmente* uguali se avevano gli stessi elementi, ma due funzioni erano anche *intenzionalmente* uguali se erano date dalla stessa formula.
Quando diciamo che le funzioni sono date da formule, non è necessario conoscere il dominio o il codominio della funzione. Consideriamo la funzione $f(x) = x$. Possiamo considerarla come una funzione $f: X \rightarrow X$ per ogni elemento di X.
Nella programmazione, questa visione di funzioni come regole è molto appropriata: molti programmatori non considerano infatti solo i risultati di un programma, ma conoscono anche come il risultato del programma è stato calcolato, quando tempo impiega, quando spazio usa, ecc.

Lambda calcolo

Il lambda calcolo è una teoria delle funzioni come formule. Esso è un sistema che ci permette di manipolare funzioni ed espressioni.
Il lambda calcolo usa il linguaggio di espressioni aritmetiche, formato da variabili (x,y,z…), numeri (1,2,3,…) e operatori (+,-,×,…). Un'espressione come $x + y$ da il risultato di un'addizione. Il grande vantaggio di questo linguaggio è che le espressioni possono essere annidate senza la necessità di menzionare l'esplicito risultato intermedio.
Es. scriviamo $A = (x + y) \times z^2$, e non fai $w = x + y$, poi fai $u = z^2$, poi fai $A = w \times u$, la quale è un'espressione difficile da manipolare.

Il lambda calcolo definisce un insieme di regole di riscrittura che determinano in maniera precisa come i termini stessi possano essere riscritti. In questo modo, il processo di riscrittura diventa un vero e proprio calcolo.

Il lambda calcolo estende l'idea del linguaggio delle espressioni includendo le funzioni. Quando noi normalmente scriviamo: fai che f sia la funzione $x \to x^2$, poi considera $A = f(5)$ col lambda calcolo basta scrivere $A = (\lambda x x^2)(5)$.
L'espressione $\lambda x x^2$ indica la funzione che va da x a x^2. Come in aritmetica, possiamo usare le parentesi per i gruppi di termini.
È evidente che la variabile x è la variabile attuale dei termini $\lambda\, x x^2$. Dunque non fa differenza se noi scriviamo invece $\lambda y. y^2$. Una variabile attuale è anche chiamata *variabile diretta*.
Un vantaggio della notazione del lambda calcolo è che ci permette di parlare facilmente riguardo le funzioni di ordine alto, ad esempio con le funzioni i quali elementi sono anch'essi funzioni.
Es. $f \to f \circ f$ in lambda calcolo diventa $\lambda f. \lambda x. f\big(f(x)\big)$

Termini

Il lambda calcolo, così come la Teoria delle Funzioni Ricorsive e la Macchina di Turing, è un formalismo che consente di definire il lato meccanico delle funzioni, ovvero proprio quelle procedure che consentono di produrre dei valori in uscita a partire da certi valori in ingresso.
Il lambda calcolo è un linguaggio formale le cui espressioni sono chiamate lambda termini. Chiamiamo *lambda termine* qualunque stringa generata dalla seguente grammatica: $T ::= Id | \lambda Id. T | (T\ T)$
dove Id appartiene a un insieme numerabile e infinito di variabili.
Un lambda termine può dunque essere, rispettivamente alla grammatica, un nome di **variabile**, l'**astrazione** di un termine rispetto ad una variabile o l'**applicazione** di un termine come argomento (o parametro) di un altro.
Convenzioni per le semplificazioni delle parentesi:

- M N invece di (M N)
- M N P invece di (M N)P
- $\lambda x. M\ N$ invece di $\lambda x. (M\ N)$
- $\lambda xyz. M$ invece di $\lambda x. \lambda y. \lambda z. M$

Regole di riscrittura

Definiamo come i lambda termini possono essere *riscritti*.

- **Variabili**: dato un generico lambda termine T, chiamiamo *Var(T)* l'insieme che contiene tutte le variabili menzionate in T; Chiamiamo *Libere(T)* l'insieme che contiene le sole *variabili libere* di T; Chiamiamo infine *Legate(T)* l'insieme con tutte le *variabili legate*.
  - L'insieme di tutte le variabili semplicemente raccoglie tutti i nomi di variabile presenti in un lambda termine
  - L'insieme delle variabili libere è così definito:
    1) $Libere(x) = \{x\}$
    2) $Libere(\lambda x M) = Libere(M) - \{x\}$
    3) $Libere(M\ N) = Libere\ (M) \cup Libere(N)$
  - L'insieme delle variabili legate è composto da quelle variabili che non sono libere, e quindi si ottiene come semplice differenza tra i due insiemi sopra definiti: $Legate(T) = Var(T) - Libere(T)$

  Un *nome di variabile* si dice ***fresco***, relativamente ad un termine, se esso non è compreso tra i nomi di variabile di quello stesso termine.

  Es. $Var\big(\lambda z. \lambda x. (xy)\big) = \{z, x, y\}$
  $Libere\big(\lambda z. \lambda x. (xy)\big) = \{y\}$
  $Legate\big(\lambda z. \lambda x. (xy)\big) = \{x, z\}$

- **Sostituzione***:* Chiamiamo sostituzione il rimpiazzo di tutte le occorrenze di un sotto-termine con un altro, all'interno di un terzo termine che rappresenta il contesto della sostituzione stessa. Indichiamo con: $T_1[T_2/T_3]$ la sostituzione del termine T_2 al posto di T_3 all'interno del termine T_1, il quale funge da contesto.
  Es. $(zx)[\lambda x. x/z] \equiv \big((\lambda x. x)x\big)$

  Una definizione ricorsiva dell'algoritmo di sostituzione è la successiva:
  1) $x[N/x] \equiv N$
  2) $y[N/x] \equiv y$ se $x \neq y$
  3) $(\lambda y. M)[N/x] \equiv \lambda y. M$
  4) $(\lambda y. M)[N/x] \equiv \lambda y. (M[N/x])$ se $y \neq x$ e $y \notin FV(N)$
  5) $(\lambda y. M)[N/x] \equiv \lambda z. (M[z/y][N/x])$ se $y \neq x, y \in FV(N)$
  6) $(M_1 M_2)[N/x] \equiv (M_1[N/x])\ (M_2[N/x])$

  ATTENZIONE: La sostituzione avviene solo nelle variabili libere! Non in quelle legate (ovvero con λ)

  Es. $(zx)[(\lambda x. x)/z] = ((\lambda x. x)x)$;
  $\big(\lambda z. (zx)\big)[(\lambda x. x)/z] = \big(\lambda z. (zx)\big)$;
  $\big(\lambda z. (zx)\big)[(zx)/x] = \lambda w. ((zx)[w/z][(zx)/x]) = \lambda w. \big((wx)[(zx)/x]\big) = \lambda w. w(zx)$ dove w è un nome fresco

- **Alpha conversione**: $\lambda x. M \Rightarrow_\alpha \lambda y. (M[y/x])$
  L'alfa conversione si applica ai termini che sono *astrazioni*. Data un'astrazione, possiamo riscriverla sostituendo la variabile astratta (x) con un'altra (y), a patto che, nell'intero sotto-termine, al posto di ogni occorrenza della prima, noi andiamo a scrivere la seconda.
  La regola di Alfa Conversione non si occupa di fare alcuna distinzione fra occorrenze libere o legate delle variabili, dato che l'operazione di sostituzione si occupa già di fare ciò.
  Es. $\lambda\, x. (xy) \Rightarrow_\alpha \lambda z. (zy)$
  $\lambda x. \big(x\big(\lambda z. (zw)\big) \Rightarrow_\alpha \lambda z. \Big(z\big(\lambda z. (zw)\big)\Big)$

- **Beta riduzione**: $(\lambda x.M)N \Rightarrow_\beta M[N/x]$
  La beta riduzione è analoga all'operazione di applicazione di una funzione matematica ai suoi argomenti: il termine sulla destra rappresenta l'argomento (o parametro), mentre il termine più a sinistra rappresenta la funzione stessa che viene applicata.
  Un termine per essere beta-ridotto deve essere una lambda-astrazione.
  Redex: sono i termini della forma $(\lambda x.M)N$

  In pratica dato un termine $(\lambda x.x)(y)$ per applicare una beta riduzione basta togliere il termine astratto ed applicare il termine redex alla variabile libera.
  Es. $(\lambda x.x)(y) \Rightarrow_\beta y$
  $(\lambda x.xx)(y) \Rightarrow_\beta yy$
  $\big(\lambda x.(xx)\big)(\lambda y.y) \Rightarrow_\beta (\lambda y.y) \Rightarrow_\beta \lambda y.y$
  $\big(\lambda xy.(xy)\big)\big(\lambda x.(xx)\big)(\lambda y.y) \Rightarrow_\beta \big(\lambda x.(xx)\big)(\lambda y.y) \Rightarrow_\beta \dots \Rightarrow_\beta \lambda y.y$

Forme normali
Diciamo che un termine del lambda calcolo si trova in *forma normale* se esso *non è riscrivibile* per mezzo della regola di Beta-riduzione.
Nel lambda calcolo, quindi, una *funzione calcolabile* è rappresentata da una qualche lambda espressione in grado di riscriversi fino a raggiungere un termine in forma normale.

Viceversa, esistono termini che generano una successione infinita di riscritture senza mai raggiungere una forma normale, che perciò rappresentano *funzioni non calcolabili*. Un esempio di espressione non calcolabile è la seguente:

$$\lambda y.(yy)\lambda y.(yy) \Rightarrow_\beta \lambda y.(yy)\lambda y.(yy) \Rightarrow_\beta \dots$$

Poiché il termine $D =_{def} \lambda y.(yy)$ non fa altro che prendere un termine e scriverne due copie, esso è spesso noto col termine di *duplicatore*.

Logicamente, un termine in forma normale non contiene dei redex. Esistono termini che non possiedono una forma normale:

$$\Omega \coloneqq (\lambda x.xx)(\lambda x.xx)$$

Teorema di Church e Rosser
$M \Rightarrow_\beta N$ se e solo se $\exists W.M \Rightarrow_\beta W \Leftarrow_\beta N$

Programmare in lambda calcolo
Una delle migliori caratteristiche del lambda calcolo è che può essere usato per elaborare dati come dei termini booleani e dei numeri naturali.

Termini booleani
- $T = \lambda xy.x$
- $F = \lambda xy.y$

Naturalmente valgono le solite regole dell'and (TT = T, TF = F, FT = F, FF = F), indicato con $\lambda ab.abF$.
Si può anche definire il termine *if_then_else* con $\lambda x.x$.

Numeri naturali
Se f e x sono lambda termini, e $n \geq 0$ è un numero naturale, scriviamo $f^n x$ per indicare $f\big(f(\dots(fx)\dots)\big)$ dove *f* si ripete n volte. Per tutti i numeri naturali n, definiamo il lambda termine $\bar{n}$, chiamato **numerale di Church**, dato da $\bar{n} = \lambda fx.f^n x$.
Ecco alcuni numerali di Church:

$\bar{0}$ = $\lambda fx.x$
$\bar{1}$ = $\lambda fx.fx$
$\bar{2}$ = $\lambda fx.f(x)$
$\bar{3}$ = $\lambda fx.f\big(f(fx)\big)$
…

Questo particolare metodo di elaborazione dei dati è dovuto ad Alonzo Church, l'inventore del lambda calcolo.

La *funzione di successione* può essere definita come segue: $succ = \lambda nfx.f(nfx)$. Applicato ad un numero avviene che:

$succ\ \bar{n}$ = $\big(\lambda nfx.f(bfx)\big)(\lambda fx.f^n x)$
$\rightarrow_\beta$ $\lambda fx.f\big((\lambda fx.f^n x)fx\big)$
$\rightarrow_\beta$ $\lambda fx.f(f^n x)$
= $\lambda fx.f^{n+1}x$
= $\overline{n+1}$

È possibile anche definire l'addizione e la moltiplicazione:
addizione = $\lambda nmfx.nf(mfx)$
moltiplicazione = $\lambda nmf.n(mf)$

Rappresentabilità di funzioni definitive ricorsivamente
La *tesi di Church* dice che nel lambda-calcolo possiamo rappresentare qualsiasi funzione che sia calcolabile tramite un algoritmo. Questo però potrebbe risultare non così semplice quando consideriamo funzioni, come il fattoriale, che sono definite tramite un algoritmo ricorsivo.
Sappiamo che l'algoritmo per il calcolo del fattoriale si rappresenta tramite il seguente programma funzionale:

fact n = if (n=0) then 1 else n*fact(n-1)

Sappiamo che if_then_else si può rappresentare, come anche il predicato di uguaglianza, i numeri, il prodotto e la funzione predecessore. Quindi il fattoriale si dovrebbe rappresentare con il seguente lambda-termine:

\n.**if_then_else** (**iszero** n 0) 1 (**mult** n (*fact*(**pred** n)))

Dove però al posto di fact dovremmo inserire un lambda-termine speciale chiamato *Operatore di Punto Fisso*. Utilizzando tali operatori si possono rappresentare funzioni definite ricorsivamente. Il fattoriale verrebbe infatti rappresentato dal lambda-termine

Y(\f.\n.if_then_else (iszero n 0) 1 (mult n (f(predn)))

Teorema di Church-Rosser e unicità delle forme normali

$$\forall M,P,Q: M \rightarrow_\beta P \ e \ M \rightarrow_\beta Q, \exists\, R : P \rightarrow_\beta R \ e \ Q \rightarrow_\beta R$$

Una immediata conseguenza di questo teorema è il seguente corollario: Se un termine ha una forma normale, questa è unica.

Dimostrazione: Supponiamo, per assurdo, che esista un termine M che abbia due forme normali. Tale cioè che esistano due termini in forma normale N_1 e N_2 tali che $M \rightarrow_\beta N_1 \ e \ M \rightarrow_\beta N_2$. Per il Teorema di Church Rosser deve allora esistere un termine R tale che $N_1 \rightarrow_\beta R \ e \ N_2 \rightarrow_\beta R$.

Siccome però N_1 e N_2 sono in forma normale, se $N_1 \to_\beta R$ deve necessariamente essere che arrivo ad R partendo da N_1 con zero passi di riduzione, cioè R è identico a N_1! Lo stesso discorso per $N_2 \to_\beta R$. Questo significa che N_1 e N_2 sono termini identici, e quindi non è vero che M possa avere due forme normali distinte.

Il teorema di Church-Rosser può essere formulato in modo equivalente nel seguente modo:

$$Se\ M \to_\beta N, \qquad \exists R: \quad M \to_\beta R\ e\ N \to_\beta R$$

Consistenza della teoria della beta-equivalenza

Se prendiamo ad esempio i lambda termini x e y.y, non è vero che $x \to_\beta y.y$ è un giudizio valido, altrimenti dovrebbe esistere un termine R tale che $x \to_\beta R$ e $y.y \to_\beta R$, ma questo è impossibile.

Il formalismo delle funzioni primitive ricorsive e parziali ricorsive

La classe F_μ delle funzioni $\mu - ricorsive\ o\ parziali\ ricorsive$, è definita induttivamente secondo Kleene come segue (notazione: f^k indica una funzione da $\mathbb{N}^k$ in $\mathbb{N}$):

1. *Funzione base*: le funzioni $S^1, C_h^k, P_j^k \in F_\mu, \forall\ h, k\ e\ j$.
   a. S è la funzione successione, cioè $S: N \to N: n \to n+1$
   b. C_h^k è la funzione costante di valore h, cioè $C: \mathbb{N}^k \to N: (n_1, \dots, n: k\ ) \to h$
   c. P_j^k è la funzione proiezione sulla j-esima componente, cioè $P: N^k \to N: (n_1, \dots, n_k) \to n_j$.

2. *Composizione*: se $h^m \in F_\mu$ e $\forall\ 1 \le i \le m, g_i^k \in F_\mu$, allora la funzione f^k definita come
   $$f(x_1, \dots, x_k) = h^m\left(g_1^k(x_1, \dots, x_k), \dots, g_m^k(x_1, \dots, x_k)\right)$$
   appartiene ad F_μ.

3. *Ricorsione primitiva:* se $h^{k+2}, g^k \in F_\mu$, allora la funzione f^{k+1}, definita come
   $$\begin{cases} f(0, x_1, \dots, x_k) = g(x_1, \dots, x_k) \\ f(y+1, x_1, \dots, x_k) = h(y, f(y, x_1, \dots, x_k), x_1, \dots, x_k) \end{cases}$$
   appartiene ad F_μ.
4. *Minimizzazione*: se $h^{k+1} \in F_\mu$, allora la funzione f^k, definita come
   $$f(x_1, \dots, x_k) = \mu\{y | h(y, x_1, \dots, x_k) = 0\}$$
   appartiene ad F_μ.

5. Nient'altro appartiene ad F_μ.

La classe delle *funzioni primitive ricorsive* è definita analogamente con l'esclusione della clausola di minimizzazione 4.

Sia $R \subseteq \mathbb{N}^k$

i. R è *ricorsiva* se e solo se la sua funzione caratteristica ξ_R è totale ricorsiva.
ii. R è *ricorsivamente enumerabile* se e solo se ξ_R è parziale ricorsiva, ovvero se vale che
$$\xi_R(n_1, \dots, n_k) = \begin{cases} 1 & se\ n_1, \dots, n_k \in R \\ \perp & se\ n_1, \dots, n_k \in R \end{cases}$$

Teoria della ricorsività

La teoria della ricorsività è lo studio delle funzioni, e il suo scopo è di classificarle dal punto di vista della loro difficoltà di calcolo.
Una prima classificazione consiste nel separare le **funzioni ricorsive** (cioè quelle calcolabili mediante un computer) dalle altre.

Funzioni ricorsive

I progenitori della teoria sono *Babbage*, che formulò quella che chiamiamo la Tesi di Church, ovvero l'asserzione che la nozione di funzione ricorsiva è un equivalente matematico preciso della nozione informale di funzione calcolabile; e *Dedekind*, il quale notò che i numeri naturali sono generati a partire dallo 0 mediante l'operazione successore S, tale che $S(x) = x + 1$. Poiché i numeri sono tutti della forma 0 o $S(x)$, Dedekinf introdusse il principio di definizione per **ricorsione primitiva**: per definire una funzione su tutti i numeri naturali è sufficiente stabilire il suo valore per 0, e descrivere come si può passare dal valore per x al valore per S(x).
Ackermann scoprì una funzione facilmente calcolabile che non è ricorsiva primitiva, detta **diagonalizzazione**: si possono enumerare le funzioni ricorsive primitive di un argomento mediante combinazioni di ricorsioni primitive, mettendole in ordine alfabetico; si ottiene così una lista $f_0\ f_1\ f_2$ ... di tutte le funzioni ricorsive di un argomento.
Numerosi furono i tentativi per trovare quel fosse la classe delle funzioni calcolabili, fino a quando Kleene diede inizio ad una serie di risultati che culminarono nella seguente sorpresa: tutte le definizioni proposte sono equivalenti e descrivono sempre la classe delle funzioni ricorsive. Il metodo usato per dimostrare tale equivalenza è detto **aritmetizzazione** e consiste nell'assegnare numeri a oggetti in modo sistematico ed effettivo, e nel tradurre proprietà degli oggetti in proprietà dei loro corrispondenti numeri.

Tesi di Church

Nel 1936 Church e Turing enunciarono la cosiddetta Tesi di Church: *le funzioni ricorsive sono esattamente le funzioni calcolabili*.
La teoria della ricorsività fornisce uno strumento per provare matematicamente limitazioni delle possibilità del pensiero umano.
Fra le varie definizioni equivalenti di funzione ricorsiva abbiamo la calcolabilità mediante macchine astratte, le cosiddette macchine di Turing: esse sono i computers, astratti dalle loro limitazioni fisiche (possibilità di rotture, limite di memoria). Qualunque computer è in grado di calcolare ogi funzione ricorsiva di cui gli sia fornito un programma. La tesi di Church asserisce che le sole funzioni per cui esistono programmi sono le funzioni ricorsive, e dunque che abbiamo raggiunto i limiti della potenza di calcolo meccanico: sarà possibile in futuro migliorare l'efficienza dei computers ma non la loro potenza assoluta. In particolare, così come già per il pensiero umano, la teoria della ricorsività fornisce uno strumento per provare matematicamente limitazioni della possibilità dei computer.

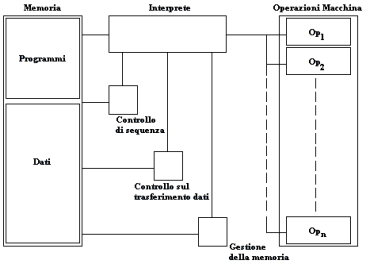
Computers e linguaggi di programmazione

Come è noto, i computers calcolano funzioni mediante programmi scritti in linguaggi di programmazione, ci cui esiste una grande varietà. La teoria della ricorsività fornisce una spiegazione di tale varietà: poiché i computers calcolano tutte e sole le funzioni ricorsive, ogni definizione equivalente di ricorsività descrive un approccio alternativo alla calcolabilità attraverso computers, e dunque un tipo di linguaggio di programmazione.
La dimostrazione di equivalenza fra un dato approccio e la calcolabilità mediante computers consiste nel tradurre il metodo di calcolo di una generica funzione implicito nel dato approccio in una serie di istruzioni eseguibili direttamente dal computer (il cosiddetto linguaggio macchina). Tale dimostrazione consiste nella costruzione di un **compilatore**.

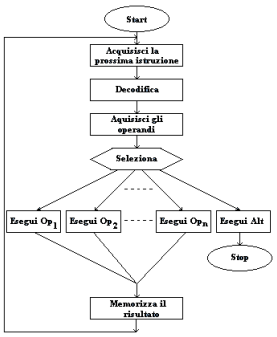
Macchine astratte

Una macchina astratta (Imperativa) è un insieme di strutture dati e algoritmi in grado di memorizzare ed eseguire programmi. Le componenti principali di una macchina astratta sono:

- Un *interprete;*
- Una *memoria*, destinata a contenere il programma che deve essere eseguito e i dati su cui si sta operando;
- Un insieme di *operazioni primitive* utili all'elaborazione dei dati primitivi;
- Un insieme di operazioni e strutture dati che gestiscono il *flusso di controllo*, ovvero che governano l'ordine secondo il quale le operazioni e le istruzioni, descritte dal programma, vengono eseguite;
- Un insieme di operazioni e strutture dati per il controllo del *trasferimento dei dati*, che si occupa di recuperare gli operandi e memorizzare i risultai delle varie istruzioni;
- Un insieme di operazioni e strutture dati per la *gestione della memoria*.



Le varie macchine astratte differiscono per il diverso modo di gestire l'esecuzione di un programma.



Interprete

La componente fondamentale che dà alla macchina astratta la capacità di eseguire programmi è l'interprete: esso coordina il lavoro delle altre parti della macchina astratta eseguendo un semplice ciclo FETCH/EXECUTE, finché non viene eseguita una particolare istruzione primitiva (detta HALT) che ne provoca l'arresto immediato.
Il processo eseguito dall'interprete è suddiviso in fasi:

- Inizialmente avviene l'acquisizione, mediante l'uso delle strutture dati preposte al controllo della sequenza, della prossima istruzione da eseguire (FETCH);
- L'istruzione viene decodificata e si acquisiscono i suoi eventuali operandi, ricorrendo al controllo sul trasferimento dei dati;
- Viene eseguita l'opportuna operazione primitiva e l'eventuale risultato viene memorizzato;
- Se l'operazione eseguita non e' quella che fa arrestare l'interprete, il processo ricomincia.

Esempio di macchina astratta

La nozione di macchina astratta è ben più generica di quanto si possa credere. Infatti anche un *ristorante*, in un certo senso si può vedere come una macchina astratta. I programmi "*eseguiti*" da tale macchina sono composti da sequenze di ordinazioni di piatti (Es.: antipasto alla marinara; linguine al pesto; pepata di cozze; macedonia; limoncello). Supponiamo di avere un ristorante in cui ci sia un cuoco specializzato per ogni piatto ed un insieme di inservienti preposti a portare a tali cuochi gli ingredienti per i loro piatti. L'insieme dei cuochi può esser visto come l'insieme delle "*operazioni*" della macchina. La "*memoria*" della nostra macchina sarà il taccuino del cameriere. L'"*interprete*" del ristorante può essere il cameriere stesso che, letta la prima "istruzione" la "decodifica", dicendo agli inservienti quali ingredienti portare a quale cuoco. Ovviamente anche la dispensa dovrà esser vista come parte della memoria del ristorante (la parte che memorizza gli "argomenti" delle istruzioni). Un altro cameriere provvederà a "*memorizzare*" sul nostro tavolo il risultato dell'esecuzione dell'istruzione.

Linguaggi di programmazione

Una macchina astratta esegue programmi memorizzati al suo interno. Il *linguaggio macchina* (LM) di una macchina astratta M è il linguaggio in cui si esprimono tutti i programmi interpretabili dall'interprete di M. LM definisce quindi l'insieme delle strutture dati che realizzano la rappresentazione interna dei programmi eseguibili da M.
Spesso però è più conveniente pensare ad un programma in termini di stringhe di caratteri. L'insieme di queste versioni dei programmi forma il cosiddetto *LMEST* (Linguaggio Macchina ESTerno).
Il programma espresso in forma direttamente memorizzabili nella macchina e la sua versione mnemonica sono due rappresentazioni dello stesso oggetto: il programma astratto. Quindi useremo il termine linguaggio macchina ad indicare indifferentemente LM o LMEST.
Es. LM = i vari livelli di tensione nei bit di una memoria fisica
LMEST = un insieme di stringhe di caratteri "0" e "1"
Il compito di realizzare la conversione dal linguaggio LMEST al linguaggio LM è facilmente automatizzabile ed è svolto dal LOADER (così detto perchè carica il programma nella memoria della macchina astratta).
Così come data una macchina astratta resta definito un linguaggio di programmazione (il suo linguaggio macchina), analogamente dato un linguaggio di programmazione resta definita una macchina astratta che ne supporta le caratteristiche principali e che ha il dato linguaggio come suo linguaggio macchina.
È da notare che *gli High Level Language* o *HLL* (cioè quei linguaggi che consentono al programmatore di esprimere i propri algoritmi in una forma molto vicina a quella in cui li ha pensati) definiscono macchine astratte tanto più complesse quanto maggiore è la potenza espressiva del linguaggio in questione. Di conseguenza la realizzazione in hardware è opportuna solo per macchina astratte relativamente semplici, mentre nel caso di una macchina astratta associata ad un linguaggio di alto livello questa scelta è poco conveniente, e risulta preferibile una realizzazione non hardware.

Realizzazione di macchine astratte

La differenza di potenza espressiva fra una macchina astratta che vogliamo realizzare e la macchina che abbiamo a disposizione per tale realizzazione (*Macchina Ospite* o HOST) è detta **semantic gap**.
Spesso il **semantic gap** tra la macchina da realizzare e la macchina HOST è talmente grande che è opportuno introdurre una o più macchine astratte intermedie.
Tipicamente, nell'implementare un linguaggio di programmazione (cioè la sua corrispondente macchina astratta) non si procede mai per pura compilazione o pura interpretazione su una data macchina HOST. La pura interpretazione potrebbe non essere soddisfacente a causa della scarsa efficienza della macchina realizzata emulando via software strutture dati, algoritmi e soprattutto l'interprete; mentre la compilazione a causa di un notevole semantic gap potrebbe portare a produrre programmi per la macchina ospite di dimensioni eccessive e magari lenti, senza considerare le difficoltà che si possono incontrare per sviluppare dei traduttori tra linguaggi eccessivamente diversi.
Costruiremo dunque la nostra macchina così (figura a sinistra).

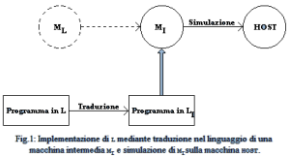


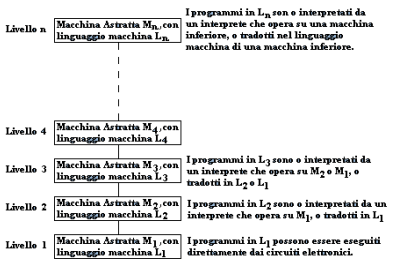
Fig. 1: implementazione di L mediante traduzione nel linguaggio di una macchina intermedia M_L e simulazione di M_L sulla macchina ospr.

Per colmare il divario fra la macchina ML e la macchina HOST, si progetta un'apposita macchina intermedia M_I che viene realizzata sulla macchina HOST. A questo punto la traduzione dei programmi in L avverrà in termini del linguaggio di M_I, e successivamente il programma per M_I così ottenuto verrà eseguito tramite interpretazione sulla macchina HOST.
Nel progetto della macchina intermedia bisogna tenere conto di due requisiti di efficienza:

1. velocità di simulazione della Macchina Intermedia M_I sulla macchina HOST;
2. compattezza del codice prodotto nel linguaggio della macchina M_I.

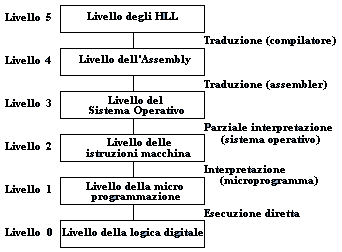
Per soddisfare tali requisiti, tipicamente M_I è solo un'estensione della macchina HOST, nel senso che ne condivide l'interprete, e ne potenzia alcuni aspetti mediante un insieme di routine note come *Run-Time System*, atto a colmare, negli aspetti, la differenza delle due macchine.

Es. Un esempio concreto di applicazione di questo schema si ha per il linguaggio di programmazione Java.



In questo caso, la macchina intermedia M_i è la JVM (*Java Virtual Machine*), il cui linguaggio è detto **Bytecode**, mentre il Run-Time System viene detto *Java RunTime Enviroment*. Per quanto riguarda la macchina HOST , nella pratica si hanno diverse realizzazione: dal Pentium II all'UltraSPARC, al PicoJavaII.
Gli scopi di tale stratificazione sono molteplici: gestire la complessità di progettazione, aumentare la flessibilità del sistema ecc.
Il livello più basso rappresenta il computer reale e il linguaggio macchina che esso è in grado di eseguire direttamente. Ciascuno dei livelli superiori rappresenta una macchina astratta, i cui programmi devono essere o tradotti in termini di istruzioni di uno dei livelli inferiori (non necessariamente del livello immediatamente al di sotto), o interpretati da un programma che gira su di una macchina astratta di livello strettamente inferiore.

Organizzazione a livelli dei sistemi di calcolo



Un tipico computer moderno si può quindi pensare come una serie di macchine astratte realizzate una sopra l'altra, ciascuna in grado di fornire funzionalità via via più potenti.
Per chiarire le idee sulla complessità di questa decomposizione in livelli, esaminiamo una possibile stratificazione dei comuni PC (figura a destra).

- *Livello 0*: Le componenti fondamentali del *livello 0* sono le cosiddette porte logiche (GATES): a partire da raggruppamenti di esse si formano i registri e le memorie. Le GATES sono dispositivi elettronici che, pur appartenendo al mondo analogico, costituiscono il ponte verso il mondo digitale, poiché esse tipicamente realizzano funzionalità (Not, And, Or, ...) che sono ben formalizzabili in termini di una teoria discreta nota come **Algebra di Boole**.
- *Livello 1*: Collezioni di registri e circuiti digitali che realizzano funzionalità aritmetico-logiche sono alla base del *livello 1*. Qui troviamo altri elementi di interesse come il datapath e le strutture che presiedono al suo controllo.
  A questo livello inizia a diventare chiara l'idea di **flusso di informazione** poiché sequenze di bit viaggiano da una componente all'altra (della macchina astratta di questo livello) venendo eventualmente elaborate.
- *Livello 2*: è quello cui tipicamente ci si riferisce quando si parla di **linguaggio macchina** di un certo computer; tipiche istruzioni di questo livello sono ADD, MOVE, SUB, ...
- *Livello 3*: evidenzia quella caratteristica precedentemente notata di non coprire del tutto i livelli sottostanti: si tratta del livello del **Sistema Operativo** (SO). sono inoltre presenti nuove operazioni, che consentono un livello di astrazione notevolmente più alto nell'ambito della gestione della memoria e del flusso di controllo. Tre delle principali funzionalità offerte dal Sistema Operativo sono:
  - il concetto di *FILE*: a questo livello si realizza il salto qualitativo verso una memoria molto strutturata e organizzata gerarchicamente (**File System**) in cui informazioni complesse possono essere immagazzinate e recuperate con grande facilità
  - la *MEMORIA VIRTUALE*: il Sistema Operativo offre ai livelli di astrazione superiore una macchina astratta con una quantità di memoria principale di gran lunga superiore a quella effettiva, liberando così il programmatore dai limiti fisici derivanti dalla particolare dotazione della macchina;
  - l'astrazione di *PROCESSO* e il *MULTITASKING*: la possibilità di avere più "pezzi" di programma (**processi**) in esecuzione indipendente e (pseudo-)parallela è una delle caratteristiche che si è andata affermando negli anni.
- *Livello 4*: segna una rottura con i livelli precedenti: si tratta del livello dell'**assembly**, il primo che presenti qualche caratteristica (sebbene elementare) dei linguaggi di programmazione moderni
- *Livello 5*: entrano in gioco gli HLL: C, Java, C++, Haskell, Prolog, etc... Si tratta del livello più vario, dove troviamo tutte le possibili soluzioni implementative, anche per macchine astratte relative allo stesso linguaggio: ad esempio, esistono varianti sia interpretate che compilate del BASIC.

Come accennato in precedenza l'avvento di Java ha portato alla nascita di un livello intermedio tra quelli del linguaggio ad alto livello e quello dell'assembly. Infatti prima di essere eseguiti, i programmi Java vengono tradotti in *bytecode*, ovvero nel linguaggio macchina della *Java Virtual Machine* (JVM), che è realizzata mediante interpretazione sui livelli sottostanti; successivamente saranno i bytecode così ottenuti che verranno eseguiti sulla JVM.

Codici e rappresentazione dell'informazione numerica

Stringhe contro numeri
Una delle differenze tra il modello computazionale delle Macchine di Turing e quello delle URM è che nel primo modello vengono manipolate stringhe di simboli, mentre nel secondo numeri naturali.
Quindi, mentre domini e codomini delle funzioni calcolate a URM sono sempre i Numeri Naturali, le macchine di Turing possono calcolare funzioni che hanno come dominio (e codominio) qualsiasi insieme i cui elementi si possano rappresentare come stringhe di caratteri. I domini (e codomini) delle funzioni calcolabili da macchine di Turing sono quindi tutti quegli insiemi per i quali sia possibile definire una funzione di codifica e decodifica.
Preso un alfabeto A, una funzione di codifica per un insieme X è una funzione iniettiva

$$cod: X \to A^+$$

Mentre la funzione di decodifica è una funzione

$$decod: A^+ \to X \cup \{errore\}$$

Tale che per ogni y appartenente ad X

$$decod(cod(y)) = y$$

Poiché i numeri naturali (che indichiamo con N) si possono codificare con stringhe sull'alfabeto {0,1}, le macchine di Turing possono lavorare anche sui numeri naturali, oltre che su molti altri domini e codomini.

Codifica e decodifica con relazioni biunivoche
Sappiamo che grazie alla definizione di cardinalità dei linguaggi, possiamo stabilire una relazione biunivoca tra A^+ e l'insieme N; tale isomorfismo ci restringe a studiare solamente le funzioni che hanno come dominio e codominio N, e ci fa osservare inoltre che se un insieme X ha una funzione di codifica su A^+ per qualche A (con relativa funzione di decodifica), allora per X esisterà pure una funzione di codifica e decodifica su $\{0,1\}^+$.
Per poter considerare solo funzioni sui naturali in teoria della ricorsività e per avere la possibilità di codificare qualsiasi insieme su $\{0,1\}^+$, non è strettamente indispensabile avere un isomorfismo tra A^+ ed N. Sarebbe sufficiente avere una funzione iniettiva tra A^+ ed N.

Isomorfismo di Cantor
Possiamo definire questa applicazione iniettiva grazie all'isomorfismo di Cantor tra coppie di numeri naturali ed i numeri naturali. Per prima cosa osserviamo che i caratteri di un alfabeto si possono ovviamente rappresentare con dei numeri naturali. Per esempio potremmo decidere di rappresentare i caratteri dell'alfabeto {a,b,g,d} con i numeri naturali 0,1,2 e 3, oppure con i numeri 8,23,15 e 4, o in altro modo.
L'isomorfismo di Cantor tra le coppie di numeri naturali ed i numeri naturali si definisce nel modo seguente.

Prendiamo una tabella in cui ci siano tutti i numeri naturali sulle righe e tutti i numeri naturali sulle colonne:

	0	1	2	3	4	…
0						
1						
2						
3						
4						
…						

Ogni casella nella tabella sarà identificata quindi da una coppia di numeri naturali. Riempiamo ora tutte le caselle della tabella con tutti i numeri naturali, procedendo per diagonali, da destra in alto a sinistra in basso. La prima diagonale è fatta da una sola casella, che riempiamo con 0, poi riempiamo la seconda, la terza e così via per tutte le diagonali. Consideriamo quindi l'intera tabella infinita come se fosse stata riempita tutta in questo modo.

	0	1	2	3	4	…
0	0	1	3	6	10	…
1	2	4	7	11	16	…
2	5	8	12	17	23	…
3	9	13	18	24	31	…
4	14	19	25	32	…	…
…	…	…	…	…	…	…

Poiché ogni casella è individuata da una coppia di numeri naturali e ogni casella contiene un numero naturale diverso da quello di tutte le altre caselle, ecco pronta la nostra codifica iniettiva e suriettiva su N delle coppie di numeri naturali: una coppia (n,m) viene codificata con il numero naturale contenuto nella casella identificata dalla riga n e dalla colonna m.

Isomorfismo di Cantor per definire una funzione iniettiva tra A^+ ed N

Per prima cosa osserviamo che i caratteri di un alfabeto si possono ovviamente rappresentare con dei numeri naturali. Per esempio potremmo decidere di rappresentare i caratteri dell'alfabeto {a,b,g,d} con i numeri naturali 0,1,2 e 3, oppure con i numeri 8,23,15 e 4, o in altro modo. Se quindi consideriamo l'alfabeto {a,b,g,d} i cui caratteri decidiamo di rappresentare con i numeri naturali 0,1,2 e 3, come è possibile associare in modo iniettivo un numero alla stringa "agaa"?
Facile, prendiamo i numeri che rappresentano i primi due caratteri: 0 e 2, e rappresentiamo con il metodo di Cantor la coppia (0,2). Tale rappresentazione è 3 (vedi tabella). Prendiamo ora il terzo carattere della stringa, a, corrispondente al numero 0 e consideriamo il numero corrispondente alla coppia (3,0), che e' 9. Ora, il numero corrispondente alla coppia (9,0), dove 0 rappresenta il carattere a, e' il numero 44. La nostra rappresentazione della stringa "agaa" sarà quindi la rappresentazione della coppia (54,4) dove 4 è la lunghezza della nostra stringa.
Se indichiamo quindi con $\underline{c}$ il numero che rappresenta un carattere c e con $(-,-)' : N \times N \to N$ l'isomorfismo di Cantor tra coppie di naturali e naturali, la nostra funzione iniettiva tra A^+ ed N $F: A^+ \to N$ sarà definita come segue:

$$F(c_1c_2\ldots c_n) = ((\ldots(\ldots((\underline{c_1},\underline{c_2})',\underline{c_3})'\ldots)',\underline{c_n})',n)'$$

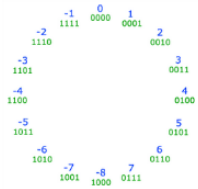
Codici

Uno dei problemi fondamentali che incontriamo quando vogliamo usare un sistema di calcolo per manipolare delle informazioni é quello di **rappresentare** le informazioni stesse all'interno del sistema. Questo problema viene risolto in informatica mediante l'adozione di opportuni **codici** che stabiliscono una corrispondenza tra la "informazione" significativa per il problema (l'applicazione) che si sta considerando ed una serie di **simboli** manipolabili dal sistema. Per esempio, i simboli manipolabili da una macchina astratta possono essere numeri interi di valore non superiore ad una soglia prefissata, quindi ogni informazione deve essere codificata sotto tale forma per poter essere trattata dal sistema.

Sia X un qualsiasi insieme (finito o infinito), che rappresenta gli oggetti o le informazioni che vogliamo trattare; Sia A un qualsiasi alfabeto finito di simboli; Con A* indichiamo l'insieme di tutte le possibili sequenze (ordinate da sinistra a destra) di simboli dell'alfabeto A, di lunghezza finita e infinita, compresa la sequenza vuota.
Definiamo una funzione iniettiva di codifica ed una relativa funzione di decodifica; L'insieme X, l'alfabeto A, e le funzioni cod e decod formano un **Codice** per la rappresentazione di elementi di X. La sequenza di simboli cod(y) corrispondente ad un qualsiasi elemento y appartenente ad X verrà chiamata "codifica di y".

Rappresentazione di numeri interi in complemento a due

Vogliamo determinare un codice a lunghezza fissa per i numeri interi. Ne esistono molti, ma il più comune è quello in **complemento alla base** (detto complemento a due). Indicheremo con *B* la base e con *p* il numero di posizioni che intendiamo utilizzare per la nostra rappresentazione a lunghezza fissa. Indicheremo con $cod_{ZcB}(-)$ la funzione di codifica degli interi in complemento alla base B, mentre con $cod_{NB}(-)$ la normale funzione di codifica posizionale dei numeri naturali.
Se B=2 e p=4 abbiamo a disposizione 16 configurazioni (indicate in verde), che possiamo disporre su di una circonferenza in modo da fornire un'idea intuitiva di come funziona la codifica dei numeri interi (indicati in blu) in complemento a due.



Formalmente, la funzione cod_{ZcB} è definita come segue:

$cod_{ZcB}(r) = cod_{NB}(r)$ se $r \geq 0$

$cod_{ZcB}(r) = cod_{NB}(B^p - |r|)$ se $r < 0$

Il numero rappresentato è negativo se e solo se il bit più a sinistra è 1.

Logica

Sistemi formali

La nozione di sistema formale corrisponde ad una formalizzazione rigorosa e completa della nozione di sistema assiomatico, che è un insieme di assiomi che possono essere usati per dimostrare teoremi.
Un sistema formale D è dato da:

- Un insieme numerabile S (alfabeto);
- Una grammatica che specifica quali sequenze finite di questi simboli sono formule ben formate, ovvero un insieme decidibile $W \subseteq S^*$ (insieme delle formule ben formate (fbf));
- Un insieme $Ax \subseteq W$ (insieme degli assiomi del sistema formale);
- Un insieme $R = \{R_i\}_{i \in I}$, con $R_i \subseteq W_i^{n_i}$ con I ed $n_i \geq 2$ finiti

La coppia $< S, W >$ è detta *linguaggio formale*.

Definizione esplicita

Dicasi *definizione esplicita* la definizione di un termine che viene aggiunto all'alfabeto del linguaggio per significare un'espressione.
Se $R \subseteq W^3$ allora scriverò $R(\alpha, \beta, \gamma)$ nella forma $\frac{\alpha \ \beta}{\gamma}$.

D-derivazione

Dato un insieme M di fbf nel sistema formale D, una D-derivazione (prova, dimostrazione) a partire da M è una successione finita di fbf $a_1, \ldots, a_n$ di D tale che, per ogni $i = 1, \ldots, n$ si abbia:

- $a_i \in Ax$ oppure
- $a_i \in M$ oppure
- $\left(a_{h_1}, \ldots, a_{h_{n_j}}\right) \in R_j \ \ per\ qualche\ j \in I, a_i = a_{h_{n_j}} e\ h_1, \ldots, h_{n_j - 1} < i$

Regole derivabili

Una formula α è derivabile nel sistema formale D a partire da un insieme di ipotesi M se e solo se esiste una D-derivazione a partire da M la cui ultima fbf è α. Scriveremo allora $M \vdash_D \alpha$ e leggeremo: M deriva α nel sistema formale D.
Se M è vuoto scriveremo $\vdash_D \alpha$ e leggeremo: α è un teorema in D se e solo se non vale $M \vdash_D \alpha$.

Regole ammissibili
Sia R l'insieme delle regole di un sistema formale D; una regola $R: \frac{a_1,\dots a_k}{a_{k+1}}, R \notin R$, è detta derivabile in D se e solo se per tutte le fbf $a_1, \dots, a_k$ che soddisfano R si ha: $a_1, \dots, a_k \vdash_D a_{k+1}$. R è detta ammissibile (o eliminabile) in D se e solo se da $\Gamma \vdash_{D\cup\{R\}} \alpha$ segue $\Gamma \vdash_D \alpha$, dove $D \cup \{R\}$ denota il sistema formale ottenuto da D con l'aggiunta della regola R.

Ogni regola derivabile è ammissibile, ma non l'inverso.
Se $M \vdash_D \alpha$ *allora esiste* $N \subseteq M$, *N finito, per il quale si ha* $N \vdash_D \alpha$. *Quindi le dimostrazioni sono esponibili in tempo finito.*
Se $M \vdash_D \alpha_1, \dots, M \vdash_D \alpha_n$ *e* $\{a_1, \dots, a_n\} \vdash_D \beta$, *allora si avrà* $M \vdash_D \beta$.

Sistema consistente
Un sistema formale D è detto consistente se e solo se esiste una $fbf\alpha$ di D tale che non esista $\vdash_D \alpha$; se D non è consistente è detto inconsistente.

Insieme delle conseguenze
Sia Γ un insieme finito di fbf di un sistema formale D; con $_D(\Gamma) = \{\alpha \in W : \Gamma \vdash \alpha\}$.

Insieme finito consistente e inconsistente
Sia Γ un insieme finito di fbf di un sistema formale D.
- Γ è detto consistente se e solo se esiste $\alpha \in W$ tale che non esista $\Gamma \vdash_D \alpha$
- Γ è detto inconsistente o contraddittorio se e solo se Γ non è consistente

Esempio di sistema formale
Chiamiamo sistema formale CL il sistema formale così definito:
- $S = \{k, s, (,), =\}$ (alfabeto);
- $W = \{P = Q | P, Q \in r\}$ dove r è l'insieme dei termini così definito:
  - $k \in r, d \in r$;
  - $se\ P, Q \in r\ allora\ (PQ) \in r$;
  - $nient'altro$ è $un\ termine$
- $Ax: per\ ogni\ P, Q, R \in r$ i seguenti sono schemi di assioma:
  - $\big((kP)Q\big) = P\ (Axk)$
  - $P = P$ (assioma di riflessività)
  - $(((sP)Q)R) = ((PR)(QR)(Axs)$
  - Nient'altro è un assioma
- $R = \{R_1, R_2, R_3, R_4\}$ dove:
  - $R_1 = \{(P = Q, Q = P) | P, Q \in r\} \subseteq W^2$ ovvero $\frac{P=Q}{Q=P}$
  - $R_2 = \{(P = Q, Q = P, P = R) | P, Q, R \in r\} \subseteq W^3$ ovvero $\frac{P=Q \quad Q=P}{P=R}$ TRANS
  - $R_3 = \{(R = R', (PR) = (QR')) | P, Q, R, R' \in r\}$ ovvero $\frac{R=R' \quad (PR)=(QR)}{(PR)=(QR')}$ CONGR1
  - $R_4 = \{(R = R', (RP) = (RQ), (RP) = (R'Q)) | P, Q, R, R' \in r\}$ ovvero $\frac{R=R' \quad (RP)=(RQ)}{(RP)=(R'Q)}$ CONGR2

Es: dimostriamo che $\vdash_{CL} \big((sk)k\big)k\big) = k$:

1. $\big((sk)k\big)k\big) = \big((kk)(kk)\big)$ Axs
2. $\big((kk)(kk)\big) = k$ Axk con $P = k$ e $Q = (kk)$
3. k $TRANS(1,2)$

Calcolo proposizionale
Chiamiamo *proposizioni* delle espressioni elementari che possono possedere un valore di verità; chiamiamo invece *variabili proposizionali* delle variabili che stanno per proposizioni: di solito le indicheremo con *p,q,r,…* Un assegnamento proposizionale è una funzione che associa ad ogni variabile proposizionale un valore di verità (0,1); chiameremo **tautologia** una fbf il cui valore di verità è sempre 1.
Il calcolo proposizionale è un sistema formale i cui teoremi sono tutte e sole le tautologie.

Sistema formale P_0
Il sistema formale P_0 è il seguente sistema formale:
- S: è formato dall'unione di un insieme numerabile di variabili proposizionali e l'insieme dei connettivi → e ¬ (implica e not) e l'insieme dei due simboli ausiliari (e);
- W: insieme delle fbf così definito:
  - Ogni variabile proposizionale è una fbf
  - Se α e β sono fbf allora lo sono ache $(\alpha \to \beta)$ e $(\neg\alpha)$
  - Nient'altro è una fbf
- Ax:
  - $\alpha \to (\beta \to \alpha)$
  - $\big(\alpha \to (\beta \to \gamma)\big) \to \big((\alpha \to \beta) \to (\alpha \to \gamma)\big)$
  - $(\neg\beta \to \neg\alpha) \to \big((\neg\beta \to \alpha\ ) \to \beta\big)$
- $R = \{MP\}$ dove MP (modus ponens) è la regola $\frac{\alpha \quad \alpha\to\beta}{\beta}$

Teorema di deduzione
Dato un insieme di $fbf\Gamma$ in P_0 si ha:

$$\Gamma \vdash_{P_0} \alpha \to \beta \ \ se\ e\ solo\ se\ \ \Gamma, \alpha \vdash_{P_0} \beta$$

Teorema di correttezza di P_0
Siano Γ un insieme di fbf di P_0 e α una fbf in P_0; si ha che:

$$se\ \Gamma \vdash_{P_0} \alpha\ allora\ \Gamma \vdash \alpha$$

P_0 è consistente.

Teorema di completezza di P_0

Sia Γ un insieme di fbf di P_0 e α una fbf di P_0; sia ha che

$$da\ \Gamma \vdash \alpha\ segue\ \Gamma \vdash_{P_0} \alpha \qquad \Gamma \vdash \alpha\ se\ e\ sole\ se\ \Gamma \vdash_{P_0} \alpha$$

La deduzione naturale per la logica proposizionale

Il sistema formale per il calcolo proposizionale descritto prima è un **sistema assiomatico** in quanto l'insieme delle regole di inferenza è composto da una sola regola: il modus ponens.

Esistono molti altri sistemi formali equivalenti a quello visto. Uno di questi è il sistema formale della deduzione naturale.

In questo sistema formale l'insieme degli assiomi è vuoto. Ci sono solo regole di inferenza.

Rispetto ai sistemi assiomatici cambia solo un poco la nozione di **derivazione**.

Infatti, nel sistema formale della deduzione naturale abbiamo che $\Gamma \vdash P$ (cioe' *"la formula P è derivabile dall'insieme di assunzioni* Γ*"*) quando è possibile costruire un **albero di derivazione di P da assunzioni in Γ**.

Calcolo proposizionale

$$v: FBF(formule\ ben\ formate) \to \{0,1\} \qquad v(A) = \begin{cases} 1\ se\ A\ \text{è}\ vera \\ 0\ se\ A\ \text{è}\ falsa \end{cases}$$

Negazione ($\neg = NOT$)	
P	¬P
0	1
1	0

Congiunzione ($\wedge = AND$)		
P	Q	P∧Q
0	0	0
0	1	0
1	0	0
1	1	1

Disgiunzione ($\vee = OR$)		
P	Q	P∨Q
0	0	0
0	1	1
1	0	1
1	1	1

Implicazione ($\rightarrow$)		
P	Q	P→Q
0	0	1
1	0	0
0	1	1
1	1	1

priorità più alta	¬
.	∧
.	∨
priorità più bassa	→

Il compito della **semantica** è assegnare un significato a tutte le frasi sintatticamente corrette (cioè a tutte le FBF).

TAUTOLOGIA: una formula ben formata P è una tautologia se P risulta vera in ogni interpretazione. In tal caso si scriverà $\models P$

Es. $A \rightarrow A \vee B$ è una tautologia:

A	B	A∨B	A→A∨B
0	0	0	1
0	1	1	1
1	0	1	1
1	1	1	1

Due formule si dicono **semanticamente equivalenti** quando hanno la stessa tavola di verità.