Elementi di Informatica Teorica

A.A. 2021-2022

Docente: A. De Luca

Dispense tratte dal corso di Informatica a cura dello studente **S. Cerrone**

1. Il concetto di calcolabilità: funzioni calcolabili e parzialmente calcolabili.

Elementi di Linguaggio S

Iniziamo con una descrizione informale del linguaggio S. Esso è composto da:

- **Variabili di Input**: un insieme potenzialmente infinito di variabili che indichiamo con i simboli $X_1, X_2, \dots, X_n$ (se si usa una sola variabile di input si può scrivere X per semplicità)
- **Variabile di Output**: un'unica variabile che rappresenta l'uscita del programma e si denota con Y
- **Variabili Locali**: interne al programma e anch'esse infinite, si usano i caratteri $Z_1, Z_2, \dots, Z_n$
- **Etichette** (label): le istruzioni nel linguaggio S sono etichettate, per le etichette si usano le lettere dell'alfabeto $A_1, \dots, A_n, B_1, \dots, B_n, C_1, \dots, C_n, D_1, \dots, D_n, E_1, \dots, E_n$ (la scelta di usare 5 lettere anziché una è per dare una migliore leggibilità al programma). Le etichette si indicano tra quadre: $[A]$
- **Istruzioni**: ne esistono solo di tre tipi e tutti lavorano su numeri naturali, un programma nel linguaggio S in generale rappresenta funzioni del tipo $f: \mathbb{N}^k \to \mathbb{N}$ (ennupla in input e numero naturale in output). Descriviamo le istruzioni nella seguente tabella (V è una qualsiasi variabile):

Incremento	$V \leftarrow V + 1$	Incrementa di 1 il valore della variabile V
Decremento	$V \leftarrow V - 1$	Decrementa di 1 il valore della variabile V, per $V = 0$ invece non fa nulla
Salto	$IF\ V \neq 0\ GOTO\ L$	Se il valore di $V \neq 0$, esegue l'istruzione etichettata L, altrimenti procede con la successiva (la condizione è fissa)

N.B.: tutte le variabili locali Z_i e la variabile di output Y partono sempre dal valore 0

L'esecuzione di un programma in S termina quando non "sa" quale istruzione eseguire, e questo accade o quando non esiste una istruzione successiva (il programma termina per EOF) oppure c'è una istruzione di salto ad una etichetta inesistente. Diamo a questo punto un esempio di semplice programma:

$[A]$ $\boldsymbol{X \leftarrow X - 1}$
$\boldsymbol{Y \leftarrow Y + 1}$
$\boldsymbol{IF\ X \neq 0\ GOTO\ A}$

Questo programma distrugge la variabile di input e restituisce come output il valore della variabile X, descrive praticamente la funzione a destra.

$$f(x) = \begin{cases} 1\ se\ x = 0 \\ x\ altrimenti \end{cases}$$

Già da questo programma è evidente che per definire funzioni complesse ci sia bisogno di molte istruzioni, per semplificare la scrittura del programma nascono le **MACRO** che praticamente ci dicono di prendere quella istruzione e sostituirla con un blocco scritto in linguaggio S (le macro in sé non appartengono al linguaggio). Andiamo a definire ora le macro più utilizzate:

- **Macro: Salto incondizionato**

  $\boldsymbol{GOTO\ A}$ e rappresenta il seguente blocco di istruzioni in S: $Z \leftarrow Z + 1$ / $IF\ Z \neq 0\ GOTO\ A$

  Grazie a questa macro possiamo scrivere la funzione identica $f(x) = x$ (sempre però distruttiva per la variabile x):

  $[A]$ $\boldsymbol{IF\ X \neq 0\ GOTO\ B}$
  $\boldsymbol{GOTO\ E}$
  $[B]$ $\boldsymbol{X \leftarrow X - 1}$
  $\boldsymbol{Y \leftarrow Y + 1}$
  $\boldsymbol{GOTO\ A}$

- **Macro: Azzeramento**

  $\boldsymbol{V \leftarrow 0}$ che rappresenta il blocco $\begin{matrix}[A]\ V \leftarrow V - 1 \\ IF\ V \neq 0\ GOTO\ A\end{matrix}$

Definiamo a questo punto la funzione $f(x) = x$ che ci dà in output il valore della variabile x senza distruggere quest'ultima:

$[A]$ $\boldsymbol{IF\ X \neq 0\ GOTO\ B}$
$\boldsymbol{GOTO\ C}$
$[B]$ $\boldsymbol{X \leftarrow X - 1}$
$\boldsymbol{Y \leftarrow Y + 1}$
$\boldsymbol{Z \leftarrow Z + 1}$
$\boldsymbol{GOTO\ A}$
$[C]$ $\boldsymbol{IF\ Z \neq 0\ GOTO\ D}$
$\boldsymbol{GOTO\ E}$ (è convenzione usare E, iniziale di exit, per terminare un programma)
$[D]$ $\boldsymbol{Z \leftarrow Z - 1}$
$\boldsymbol{X \leftarrow X + 1}$
$\boldsymbol{GOTO\ C}$

- **Macro: Copia**

  Rappresentato dalla macro $\boldsymbol{V \leftarrow V'}$ e semplicemente copia la variabile V' nella variabile V, ha come codice il seguente programma:

  $\boldsymbol{V \leftarrow 0}$
  { qui va il programma descritto precedentemente, sostituendo ad X la variabile V' e ad Y la V **}**

Definiamo ora una funzione somma $f(x_1, x_2) = x_1 + x_2$**:**

$\boldsymbol{Y \leftarrow X_1}$
$\boldsymbol{Z \leftarrow X_2}$
$[B]$ $\boldsymbol{IF\ Z \neq 0\ GOTO\ A}$
$\boldsymbol{GOTO\ E}$
$[A]$ $\boldsymbol{Z \leftarrow Z - 1}$
$\boldsymbol{Y \leftarrow Y + 1}$
$\boldsymbol{GOTO\ B}$

Se f è una funzione **calcolabile** essa può essere una macro così definita $\boldsymbol{V \leftarrow f(V_1, \dots, V_n)}$, ad esempio il programma somma può essere scritto come macro nel seguente modo: $\boldsymbol{Z_1 \leftarrow X_1 + X_2}$

Esercizi: Scrivere degli S programmi per il calcolo delle seguenti funzioni: $g(x) = 3x$ e $f(x_1, x_2) = x_1 \cdot x_2$ (si possono usare le macro precedentemente definite, compresa la macro somma).

- **Macro: Assegnazione**

  Rappresentata da $\boldsymbol{V \leftarrow n}$, con $n \in \mathbb{N}$ e rappresenta il seguente codice: $\left.\begin{matrix}V \leftarrow 0 \\ V \leftarrow V + 1 \\ \vdots \\ V \leftarrow V + 1\end{matrix}\right\} n$ volte

  N.B.: deve essere un numero naturale e non una variabile, quest'ultima infatti non potrebbe essere usata come macro poiché per come abbiamo pensato la somma si potrebbero avere problemi di conflitto, anche se in realtà basterebbe usare delle variabili di appoggio per ovviare il problema

- **Macro: Salto Generale**

  La definizione di questa macro è possibile grazie all'introduzione dei predicati (vedi paragrafo successivo): $\boldsymbol{IF\ Q(V_1, \dots, V_k)\ GOTO\ L}$ che tradotto: $\begin{matrix}Z \leftarrow Q(V_1, \dots, V_k) \\ IF\ Z \neq 0\ GOTO\ L\end{matrix}$

Concetti di Base

Scriviamo un programma che calcola la funzione $f(x_1, x_2) = \begin{cases} x_1 - x_2 \; se \; x_1 \geq x_2 \\ \uparrow \; altrimenti \end{cases}$ (la freccia ↑ indica che la funzione non è definita):

$Y \leftarrow X_1$
$Z \leftarrow X_2$
$[C]$ $IF\ Z \neq 0\ GOTO\ A$
$GOTO\ E$
$[A]$ $IF\ Y \neq 0\ GOTO\ B$
$GOTO\ A$
$[B]$ $Y \leftarrow Y - 1$
$Z \leftarrow Z - 1$
$GOTO\ C$

È evidente che il programma non termina mai per $x_1 < x_2$, una funzione che non è definita per tutti gli input è detta funzione parziale, essa fa in modo che il programma non termini mai per input sbagliati (la cosa è voluta e necessaria poiché il linguaggio S lavora sui soli numeri naturali). Di seguito definiamo i concetti di base della calcolabilità:

Funzione parziale e totale

Una funzione $f: D \subseteq \mathbb{N}^k$ si definisce **parziale**, mentre si dice **totale** se $D = \mathbb{N}^k$ (il dominio è tutto $\mathbb{N}^k$).
Poiché una funzione totale è una particolare funzione parziale chiameremo quest'ultima solo funzione.
Una funzione non definita ovunque è detta **non totale**.

Predicato

Per predicato si intende una proprietà verificabile tramite valori di verità (Vero o Falso) definita su $\mathbb{N}^k$, ad esempio il predicato di uguaglianza $P(x_1, x_2) \Leftrightarrow x_1 = x_2$. Stabiliamo che i predicati sono assimilabili a **funzioni totali** (non possono essere definite solo per una parte del dominio) e che i valori di verità sono 1 (se il predicato è vero) e 0 (se falso).
Esercizio: Scrivere un S programma che calcola il predicato di uguaglianza.

Funzione parzialmente calcolabile

Una funzione (parziale) è parzialmente calcolabile se esiste un S programma che la calcola, cioè restituisce il valore $f(x_1, \dots, x_k)$ per tutti gli input in cui è definita e non termina per gli altri.
Una funzione f si dirà **calcolabile** se è totale e parzialmente calcolabile.

Sintassi formale del linguaggio S

Oltre agli elementi precedentemente descritti di variabili, ed etichette (vedi Elementi di linguaggio S) un S programma è composto ancora da:

- **Asserzioni** (statement):
  1. $V \leftarrow V$ asserzione pigra (utile quando si assegna un numero ad ogni programma)
  2. $V \leftarrow V + 1$
  3. $V \leftarrow V - 1$
  4. $IF\ V \neq 0\ GOTO\ L$
- **Istruzione**: sarà o un asserzione oppure un asserzione etichettata.

Formalmente un S programma è una lista (ennupla) ordinata di istruzioni. Anche una lista vuota è un S programma che descrive semplicemente una funzione costantemente uguale a zero (da subito non c'è l'iterazione successiva), ovvero $f(0, \dots, 0)$.

Chiameremo **lunghezza** del programma semplicemente il numero di istruzioni.

- **Stato**: uno stato di un S programma è un insieme di uguaglianze della forma $V = n$ contenente esattamente un uguaglianza per ogni variabile che occorre nel programma. V deve avere un solo valore per ogni stato del programma (semplicemente devo sempre poter dire dopo ogni istruzione che valore hanno le variabili in quel momento)

- **Istantanea**: l'istantanea di un programma $\mathcal{P}$ è una coppia (i, σ) dove $1 \leq i \leq n + 1$ con n lunghezza di $\mathcal{P}$ e σ è uno stato di $\mathcal{P}$.
  L'istantanea si dice **terminale** se $i = n + 1$ (caso in cui il programma termina).
  Se (i, σ) non è terminale, l'istantanea **successiva** sarà (j, τ) dove:
  1. Se l'i-esima asserzione è quella pigra $(V \leftarrow V)$ si verifica che $\tau = \sigma$ e l'istruzione successiva sarà $j = i + 1$
  2. Se l'asserzione è l'incremento lo stato cambia, e τ sarà lo stato σ modificando l'uguaglianza V che in i sarà $V = n$ e in $j = i + 1$ invece $V = n + 1$
  3. Se l'asserzione è di decremento τ sarà lo stato σ modificando l'uguaglianza V che in i sarà $V = n$ e in $j = i + 1$ invece $V = n - 1$; mentre $\tau = \sigma$ se $V = n = 0$.
  4. Se l'i-esima asserzione è di salto $(IF\ V \neq 0\ GOTO\ L)$ allora lo stato del programma non cambia (poiché nessuna variabile cambia valore) $\tau = \sigma$, quello che cambia sarà j:
     - $j = i + 1$ se σ contiene $V = 0$
     - Se $\mathcal{P}$ contiene almeno una istruzione (anche più di una) etichettata L allora j sarà la prima istruzione etichettata L, altrimenti (se non esiste nessuna istruzione etichettata L), $j = n + 1$

Considerato il programma a destra, esempi di stati validi sono $\{X = 2, Y = 3\}$, oppure $\{X = 5, Y = 0, Z_3 = 7\}$; mentre stati non validi sono $\{X = 0, Z_2 = 1\}$ (poiché manca una variabile che occorre nel programma) e nemmeno $\{Y = 2, Y = 4, x = 0\}$ (poiché una stessa variabile viene ripetuta con valori diversi). Sia $\sigma = \{X = 5, Y = 0\}$, qual è l'istantanea successiva di $(1, \sigma)$?
Essendo una asserzione di decremento l'istantanea sarà $(2, \tau)$ con $\tau = \{x = 4, Y = 0\}$.

1. $[A]\ X \leftarrow X - 1$
2. $\quad Y \leftarrow Y + 1$
3. $\quad IF\ X \neq 0\ GOTO\ A$

Si dice **calcolo** per un programma $\mathcal{P}$ una successione di istantanee $S_1, \dots, S_k, \dots$ dove per ogni $i \geq 1$, S_{i+1} è l'istantanea successiva di S_i e la successione è o infinita (**calcolo non terminante**) oppure è finita e l'ultimo termine della successione è un'istantanea terminale (**calcolo terminante**).

- **Stato iniziale**: corrispondente a una k-upla $(r_1, \dots, r_k) \in \mathbb{N}^k$, che rappresenta lo stato $\{X_1 = r_1, \dots, X_k = r_k, Y = 0\} \cup \{V = 0 | V \text{ occorre in } \mathcal{P} \text{ e non è in } X_1, \dots, X_k\}$ (quindi variabili di appoggio ed eventuali input $X_{k+1}, \dots, X_n$ sono posti uguali a zero).

- **Istantanea iniziale**: è semplicemente la coppia $(1, \sigma)$ con σ stato iniziale.

Riprendiamo il programma usato precedentemente che rappresenta la funzione $f(x) = \begin{cases} 1\ se\ x = 0 \\ x\ altrimenti \end{cases}$; per $k = 1$ e $r_1 = 6$ lo stato iniziale corrispondente sarà $\sigma = \{X = 6, Y = 0\}$, per $k = 2$ e $(r_1, r_2) = (2,1)$ allora $\sigma' = \{X_1 = 2, X_2 = 1, Y = 0\}$; mentre per $k = 0$ sarà $\sigma'' = \{X = 0, Y = 0\}$ (aggiungere variabili che non occorrono nel programma come $Z_2 = 0$, seppure stato valido, non rappresenta uno stato iniziale), ne segue che istantanee iniziali degli stati descritti saranno rispettivamente $(1, \sigma)$, $(1, \sigma')$ e $(1, \sigma'')$.

Formalizzazione generale di funzioni calcolabili di un programma

Per ogni k e ogni $(x_1, \dots, x_k) \in \mathbb{N}^k$, definiamo la funzione calcolabile di un programma come:

$$\Psi_{\mathcal{P}}^{(k)}(x_1, \dots, x_k) = \begin{cases} y, & \text{se } \mathcal{P} \text{ ha un calcolo terminante che inizia con l'istantanea iniziale} \\ & \text{corrispondente a } (x_1, \dots, x_k) \text{ e l'istantanea terminale contiene } Y = y \\ \uparrow, & \text{altrimenti} \end{cases}$$

In generale, k può essere maggiore del numero di variabili di input $\mathcal{P}$, in tal caso alcuni input saranno ignorati, oppure k può essere minore e in tal caso gli ingressi "mancanti" sono posti a zero.

Una funzione parziale f di k variabili si dice **parzialmente calcolabile** se esiste un programma $\mathcal{P}$ tale che $f(x_1, \dots, x_k) = \Psi_{\mathcal{P}}^{(k)}$.

Se f è parzialmente calcolabile, la macro $\boldsymbol{V' \leftarrow f(V_1, \dots, V_k)}$ è sostituita dalle seguenti asserzioni:

$$\begin{aligned} & Z_m \leftarrow 0 \\ & Z_{m+1} \leftarrow V_1 \\ & \quad \vdots \\ & Z_{m+k} \leftarrow V_k \\ & Z_{m+k+1} \leftarrow 0 \\ & \quad \vdots \\ & Z_{m+k+h} \leftarrow 0 \\ & Q_m \\ [E_m] \quad & V' \leftarrow Z_m \end{aligned}$$

dove f è calcolata da $\mathcal{P} = \mathcal{P}(Y, X_1, \dots, X_k, Z_1, \dots, Z_h, E, A_1, \dots, A_l)$ e $Q_m = \mathcal{P}(Z_m, Z_{m+1}, \dots, Z_{m+k}, Z_{m+k+1}, \dots, Z_{m+k+h}, E_m, A_{m+1}, \dots, A_{m+l})$, con m sufficientemente grande in modo che $Z_m, Z_{m+1}, \dots, Z_{m+k}, Z_{m+k+1}, \dots, Z_{m+k+h}, E_m, A_{m+1}, \dots, A_{m+l}$ non compaiano altrove nel programma che richiama la macro (praticamente usiamo delle variabili di appoggio così da non avere conflitti tra il programma della macro e il programma in cui usiamo la macro).

Esercizi:

1. Determinare $\Psi_{\mathcal{P}}^{(1)}(r_1), \Psi_{\mathcal{P}}^{(2)}(r_1, r_2)$ e $\Psi_{\mathcal{P}}^{(3)}(r_1, r_2, r_3)$ per il seguente programma $\mathcal{P}$:

$$\begin{aligned} & Y \leftarrow X_1 \\ [A] \quad & \text{IF } X_2 = 0 \text{ GOTO } E \\ & Y \leftarrow Y + 1 \\ & Y \leftarrow Y + 1 \\ & X_2 \leftarrow X_2 - 1 \\ & \text{GOTO } A \end{aligned}$$

2. Sia $f_k(x) = k$, mostrare che per ogni $k \geq 0$, f_k è computabile.
3. Mostrare, tramite l'implementazione di un programma, che il predicato $x_1 \leq x_2$ è computabile.

2. Composizione di funzioni di più variabili, ricorsione primitiva

Per verificare che una funzione è calcolabile o meno abbiamo attuato un approccio diretto, ora vedremo un approccio più indiretto. Vediamo rispetto a quale operazioni la classe delle funzioni calcolabili è chiusa (prese due o più funzioni calcolabili ed applichiamo queste funzioni otteniamo una nuova funzione anch'essa calcolabile).

Composizione funzionale

Siano $f, g_1, \dots, g_k$ funzioni di n variabili, e sia h funzione k-aria tali che $\forall x_1, \dots, x_n, f(x_1, \dots, x_n) = h\big(g_1(x_1, \dots, x_n), g_2(x_1, \dots, x_n), \dots, g_k(x_1, \dots, x_n)\big)$ allora f si ottiene per **composizione** da $h, g_1, \dots, g_k$.
Se $h, g_1, \dots, g_k$ sono non totali, f sarà definita per tutti e soli i valori di $x_1, \dots, x_n$ tali che $g_1(x_1, \dots, x_n)\downarrow, \dots, g_k(x_1, \dots, x_n)\downarrow$ e $h\big(g_1(x_1, \dots, x_n), \dots, g_k(x_1, \dots, x_n)\big)\downarrow$ (la freccia $\downarrow$ significa "è definita").

Osservazione: Se h e le g_i sono totali lo sarà anche f

Teorema: la composizione di funzioni parzialmente calcolabili è parzialmente calcolabile.
Dimostrazione: supponiamo che f si ottenga dalle g_i e h per composizione, e che le g_i e h siano parzialmente calcolabili. Allora f è calcolato dal seguente S programma

$$\begin{aligned} \boldsymbol{Z_1} &\leftarrow \boldsymbol{g_1}(\boldsymbol{X_1}, \dots, \boldsymbol{X_n}) \\ \boldsymbol{Z_2} &\leftarrow \boldsymbol{g_2}(\boldsymbol{X_1}, \dots, \boldsymbol{X_n}) \\ &\vdots \\ \boldsymbol{Z_k} &\leftarrow \boldsymbol{g_k}(\boldsymbol{X_1}, \dots, \boldsymbol{X_n}) \\ \boldsymbol{Y} &\leftarrow \boldsymbol{h}(\boldsymbol{Z_1}, \dots, \boldsymbol{Z_k}) \end{aligned}$$

Questo programma è tale che $\Psi_{\mathcal{P}}^{(n)}(x_1, \dots, x_n) = f(x_1, \dots, x_n)$ come volevasi dimostrare.

Ad esempio, la funzione $d(x) = 2x$ possiamo dire che è calcolabile poiché si ottiene per composizione dalla funzione di moltiplicazione, dalla funzione costantemente uguale a 2 e dalla funzione identica: $d(x) = m(2, x)$ con $m(x_1, x_2) = x_1 x_2$ calcolabile e $f_2(x) = 2$ e $i(x) = x$ anch'esse calcolabili. Analogamente, $g(x) = 4x^2$ sarà calcolabile dato che $g(x) = m\big(d(x), d(x)\big)$. Inoltre, $h(x) = 4x^2 - 2x$ è parzialmente calcolabile (essendo la sottrazione una funzione non totale) dal teorema, ma dato che $4x^2 \geq 2x\ \forall x$, è anche totale e dunque calcolabile.

Ricorsione primitiva

Questo tipo di operazione si applica alle sole funzioni totali; prima di passare al caso generale cominciamo con il vederla per le funzioni unarie:

Se f è una funzione unaria totale e g binaria totale sono tali che $\begin{cases} f(0) = k \text{ con } k \in \mathbb{N} \\ \forall t \geq 0, f(t+1) = g\big(t, f(t)\big) \end{cases}$ diciamo che f si ottiene da k e g per ricorsione primitiva.

In generale, se f è una funzione totale di $n+1$ variabili diremo che f si ottiene per **ricorsione primitiva** da h e g se e soltanto se $\begin{cases} \boldsymbol{f}(\boldsymbol{x_1}, \dots, \boldsymbol{x_n}, \boldsymbol{0}) = \boldsymbol{h}(\boldsymbol{x_1}, \dots, \boldsymbol{x_n}) \\ \forall \boldsymbol{t} \geq \boldsymbol{0}, \boldsymbol{f}(\boldsymbol{x_1}, \dots, \boldsymbol{x_n}, \boldsymbol{t+1}) = \boldsymbol{g}(\boldsymbol{t}, \boldsymbol{f}(\boldsymbol{x_1}, \dots, \boldsymbol{x_n}, \boldsymbol{t}), \boldsymbol{x_1}, \dots, \boldsymbol{x_n}) \end{cases}$

Esempio: sia $f(x_1, x_2) = x_1 + x_2$, in questo caso $n = 1$ e questa funzione viene ottenuta con ricorsione primitiva da una funzione di 1 variabile e da una funzione di incremento della seconda variabile, dunque è $\begin{cases} f(x_1, 0) = x_1 = i(x_1) \\ \forall t \geq 0, f(x_1, t+1) = f(x_1, t) + 1 = g(t, f(x_1, t), x_1) \end{cases}$ dove $g(x_1, x_2, x_3) = x_2 + 1$

Teorema: se f è una funzione di $n+1$ variabili che si ottiene per ricorsione primitiva da h n-aria e g di $(n+2)$-aria calcolabili, allora f è calcolabile.

Dimostrazione: si dimostra semplicemente con il seguente programma che calcola f come funzione $(n+1)$-aria:

$$
\begin{aligned}
& \boldsymbol{Y \leftarrow h(X_1, \dots, X_n)} \\
[A]\ & \boldsymbol{IF\ X_{n+1} = 0\ GOTO\ E} \\
& \boldsymbol{Y \leftarrow g(Z, Y, X_1 \dots, X_n)} \\
& \boldsymbol{Z \leftarrow Z + 1} \\
& \boldsymbol{X_{n+1} \leftarrow X_{n+1} - 1} \\
& \boldsymbol{GOTO\ A}
\end{aligned}
$$

3. Classi PRC, funzioni ricorsive primitive

Classi PRC

Diciamo che un insieme di funzioni totali è una **classe PRC** (chiuso rispetto alla ricorsione primitiva) se:

1. **Contiene** le funzioni iniziali, ossia
   - **Funzione nulla**: $n(x) = 0$
   - **Funzione di incremento**: $s(x) = x + 1$
   - **Funzione di proiezione**: $u_i^{(n)}(x_1, \dots, x_n) = x_i$ per ogni n e $1 \leq i \leq n$ (funzione di n variabili che restituisce la i-esima)
2. È **chiuso rispetto alla composizione e alla ricorsione primitiva**,
   - cioè se $f(x_1, \dots, x_n) = h\big(g_1(x_1, \dots, x_n), \dots, g_k(x_1, \dots, x_n)\big)$ per qualche $h, g_1, \dots, g_k$ appartenenti all'insieme di funzioni totali, allora anche f appartiene all'insieme di funzioni totali e, analogamente per la ricorsione primitiva, se vale $\begin{cases} f(x_1, \dots, x_n, 0) = h(x_1, \dots, x_n) \\ \forall t \geq 0, f(x_1, \dots, x_n, t+1) = g(t, f(x_1, \dots, x_n, t), x_1, \dots, x_n) \end{cases}$ e g, h appartengono all'insieme di funzioni totali, anche f appartiene all'insieme di funzioni totali.

Ad esempio, le funzioni calcolabili costituiscono una classe PRC.

Funzioni ricorsive primitive

Una funzione si dice **ricorsiva primitiva** se si ottiene dopo un numero finito di volte da composizione e ricorsione primitiva. Per definizione le funzioni primitive sono contenute in tutte le classi PRC (quindi sono calcolabili). Eccetto per poche eccezioni, tutte le funzioni calcolabili sono primitive.

Elenchiamo ora alcuni esempi di funzioni primitive:

- **Funzione somma $\boldsymbol{h(x,y) = x + y}$** ($x, y$ sono variabili qualsiasi, non confonderle con X e Y del linguaggio S): per dire che questa funzione è ricorsiva primitiva bisogna innanzitutto scrivere la ricorsione primitiva del caso base, ovvero: $h(x,0) = x = u_1^{(1)}(x)$ (funzione iniziale). Ne segue il caso ricorsivo> $h(x, t+1) = h(x,t) + 1 = s\big(h(x,t)\big)$ dove s è la funzione di incremento (altra funzione iniziale). A questo punto, per vederla come ricorsione primitiva serve una funzione ternaria tale che per ogni $t \geq 0$ $h(x, t+1) = s\big(h(x,t)\big) = g(t, h(x,t), x)$, ed è $g(x,y,z) = s\left(u_2^{(3)}(x,y,z)\right)$ (la funzione che prende il secondo argomento di una terna). Abbiamo così ottenuto la combinazione di due funzioni iniziali che sono ricorsive primitive e quindi $h(x,y) = x + y$ è una funzione ricorsiva primitiva.
- **Funzione prodotto $\boldsymbol{m(x,y) = x \cdot y}$**: analogamente al procedimento usato per la funzione somma si ha che $m(x,0) = 0 = n(x)$ (caso base) e $m(x, t+1) = xt + x = h(xt, x)$ (nota che $xt = m(x,t)$ e che la funzione somma è ricorsiva primitiva, come visto prima) dove $h(xt, x) = g(t, m(x,t), x)$ con $g(x,y,z) = h\left(u_2^{(3)}(x,y,z), u_3^{(3)}(x,y,z)\right)$.

Esercizio: Dimostrare che la funzione fattoriale $x!$ e la funzione esponenziale x^y sono ricorsive primitive

- **Funzione decremento** $\boldsymbol{p(x) = \begin{cases} 0 & \textbf{se } x = 0 \\ x - 1 & \textbf{altrimenti} \end{cases}}$: semplicemente $\begin{matrix} p(0) = 0 \\ p(t+1) = t = u_1^{(2)}\big(t, p(t)\big) \end{matrix}$
- **Funzione di sottrazione modificata** $\boldsymbol{x \dot{-} y = \begin{cases} x - y \textbf{ se } x \geq y \\ 0 \textbf{ altrimenti} \end{cases}}$ (il caso 0 se $x < y$ è necessario poiché abbiamo bisogno di funzioni totali): $x \dot{-} 0 = x = u_1^{(1)}$ (caso base) e $x \dot{-} (t+1) = p(x \dot{-} t)$ che è già ricorsiva primitiva (funzione di decremento) e quindi abbiamo concluso.
- **Funzione valore assoluto di una differenza** $\boldsymbol{|x - y|}$: basta osservare che $|x - y| = (x \dot{-} y) + (y \dot{-} x)$ e quindi è evidente che è ricorsiva primitiva essendo combinazione di somma e sottrazione modificata.

4. Predicati calcolabili e ricorsivi primitivi

Funzione predicati ricorsivi primitivi

Diamo altre funzioni ricorsive primitive (abbiamo definito i predicati precedentemente)

- **Predicato rilevatore di zeri** $\boldsymbol{\alpha(x) = (x = 0) = \begin{cases} 1 \textbf{ se } x = 0 \\ 0 \textbf{ altrimenti} \end{cases}}$: questa funzione è ricorsiva primitiva poiché composizione di sottrazione modificata; infatti, $\forall x, \alpha(x) = 1 \dot{-} x$, siccome $1 \dot{-} x$ restituisce 1 se $x = 0$, altrimenti restituisce 0.
- **Predicato di uguaglianza** $\boldsymbol{d(x,y) = (x = y) = \begin{cases} 1 \textbf{ se } x = y \\ 0 \textbf{ altrimenti} \end{cases}}$: semplicemente $d(x,y) = \alpha(|x - y|)$.

Esercizio: dimostrare che il predicato minore o uguale ($x \leq y$) sia ricorsivo primitivo.

PRC dei predicati rispetto le operazioni booleane

Teorema: Siano P, Q predicati n-ari appartenenti alla classe PRC C. Allora $\neg P, \neg Q, P \wedge Q, P \vee Q$ appartengono a C.

Dimostrazione: Basta osservare che $\neg P$ (analogamente $\neg Q$) non è nient'altro che la negazione di P, ovvero $\neg P(x_1, \dots, x_n) = \alpha\big(P(x_1, \dots, x_n)\big)$. Mentre $(P \wedge Q)(x_1, \dots, x_n) = P(x_1, \dots, x_n) \cdot Q(x_1, \dots, X_n)$ quindi è una composizione del prodotto, infatti, sarà 1 se $P = 1$ e $Q = 1$, mentre avremo 0 nei casi restanti. Per le leggi di De Morgan possiamo scrivere $(P \vee Q)(x_1, \dots, x_n) = \neg\big(\neg P(x_1, \dots, x_n) \wedge \neg Q(x_1, \dots, x_n)\big)$. Di conseguenza tutti i predicati appartengono alla classe C, come volevasi dimostrare.

- **Predicato minore** $\boldsymbol{x < y}$: è possibile dire che sia ricorsivo primitivo (senza bisogno di calcolarlo) grazie al teorema precedente, infatti $(x < y) \Leftrightarrow \neg(y \leq x)$.

5. Definizione di funzioni per casi

Teorema: definizione per casi di due funzioni

Siano g, h funzioni n-arie appartenenti alla classe PRC C, e sia P predicato n-ario in C. Allora la funzione $f(x_1, \dots, x_n) = \begin{cases} g(x_1, \dots, x_n) \; se \; P(x_1, \dots, x_n) \text{ (se il predicato è vero)} \\ h(x_1, \dots, x_n) \; altrimenti \; \big(\text{quindi se } \neg P(x_1, \dots, x_n)\big) \end{cases}$ appartiene a C.

Dimostrazione: La funzione f è ricorsiva primitiva, ed è evidente se riscriviamo la funzione f nel seguente modo: $f(x_1, \dots, x_n) = g(x_1, \dots, x_n) \cdot P(x_1, \dots, x_n) + h(x_1, \dots, x_n) \cdot \underbrace{\alpha\big(P(x_1, \dots, x_n)\big)}_{\neg P(x_1, \dots, x_n)}$

Corollario: definizione per casi di $\boldsymbol{m+1}$ funzioni

Sia $f(x_1,\dots,x_n)=\begin{cases} g_1(x_1,\dots,x_n)\ se\ P_1(x_1,\dots,x_n) \\ \vdots \\ g_m(x_1,\dots,x_n)\ se\ P_m(x_1,\dots,x_n) \\ h(x_1,\dots,x_n)\quad altrimenti \end{cases}$ e siano $g_1,\dots,g_m,h,P_1,\dots,P_m$ appartenenti ad una classe PRC. Allora f appartiene alla stessa classe PRC.

Dimostrazione: Sfruttiamo il principio di induzione ed il teorema precedente. Definiamo $h'(x_1,\dots,x_n)=\begin{cases} g_m(x_1,\dots,x_n)\ se\ P_m(x_1,\dots,x_m) \\ h(x_1,\dots,x_n)\ altrimenti \end{cases} \in C$. Allora $f(x_1,\dots,x_n)=\begin{cases} g_1(x_1,\dots,x_n)\ se\ P_1(x_1,\dots,x_m) \\ \vdots \\ g_{m-1}(x_1,\dots,x_n)\ se\ P_{m-1}(x_1,\dots,x_m) \\ h'(x_1,\dots,x_n)\ altrimenti \end{cases}$ per induzione, supposto vero per m, si ha che $f \in C$, come volevasi dimostrare.

6. Operatori di somma e prodotto, quantificatori limitati

Sommatoria e Produttoria

Sia f funzione $(n+1)$-aria appartenente a C classe PRC. Allora le seguenti funzioni appartengono a C

$$g(y,x_1,\dots,x_n)=\sum_{t=0}^{y} f(t,x_1,\dots,x_n)=f(0,x_1,\dots,x_n)+\dots+f(y,x_1,\dots,x_n)$$

$$h(y,x_1,\dots,x_n)=\prod_{t=0}^{y} f(t,x_1,\dots,x_n)=f(0,x_1,\dots,x_n)\cdot\dots\cdot f(y,x_1,\dots,x_n)$$

Dimostrazione: per ricorsione primitiva: $\begin{matrix} g(0,x_1,\dots,x_n)=f(0,x_1,\dots,x_n) \\ g(y+1,x_1,\dots,x_n)=g(y,x_1,\dots,x_n)+f(y+1,x_1,\dots,x_n) \end{matrix}$.
Ovvero che $g \in C$, poiché composizione di funzioni ricorsive primitive. Si procede in modo analogo per la produttoria (si ha il prodotto al posto della somma) e quindi anche $h \in C$ come volevasi dimostrare.

Quantificazione esistenziale limitata e universale limitata

Sia P predicato $(n+1)$-ario in C classe PRC. Allora sono in C anche i seguenti predicati:

- $Q(y,x_1,\dots,x_n)=\exists t \le y \mid P(t,x_1,\dots,x_n)$ (quantificazione esistenziale limitata)
  $Q(y,x_1,\dots,x_n)$ è verificato $\Leftrightarrow \exists t \le y$ tale che $P(t,x_1,\dots,x_n)$ è verificato

- $R(y,x_1,\dots,x_n)=\forall t \le y \mid P(t,x_1,\dots,x_n)$ (quantificazione universale limitata)
  $R(y,x_1,\dots,x_n)$ è vero $\Leftrightarrow P(t,x_1,\dots,x_n)$ è vero per ogni $t \le y$

Dimostrazione: basti osservare che il predicato $Q(y,x_1,\dots,x_n)$ equivale $\sum_{t=0}^{y} P(t,x_1,\dots,x_n) \neq 0$ mentre $R(y,x_1,\dots,x_n)=\prod_{t=0}^{y} P(t,x_1,\dots,x_n)$.

Ogni Classe PRC è chiusa rispetto all'esistenziale e all' universale.

Esempi:

- $y|x$ (y divide x) $\Leftrightarrow \exists t \le x \mid y \cdot t = x$ essendo che la funzione prodotto è ricorsiva primitiva ed anche l'uguaglianza allora per l'operatore di quantificazione esistenziale limitata anche l'operatore di divisibilità sarà ricorsivo primitivo.

- Definiamo un predicato unario $Prime(x)$ che ci dice semplicemente se x è numero primo; quindi, essendo $Prime(x) \Leftrightarrow \forall t \le x\big(t=1 \vee t=x \vee \neg(t|x)\big)$ (gli unici divisori di x sono o uno o sé stesso) questo predicato è ricorsivo primitivo.

7. Minimalizzazione limitata e non limitata

Minimalizzazione limitata

Sia P predicato $(n+1)$-ario in C classe PRC. Definiamo una funzione $(n+1)$-aria (ricorsiva primitiva)

$$g(y, x_1, \dots, x_n) = \sum_{u=0}^{y} \left(\prod_{t=0}^{u} \alpha\big(P(t, x_1, \dots, x_n)\big) \right)$$

Supponiamo che $\exists t \leq y \mid P(t, x_1, \dots, x_n)$. Allora $g(y, x_1, \dots, x_n)$ è proprio il più piccolo $t \leq y$ tale che $P(t, x_1, \dots, x_n)$ (il predicato sia verificato), ossia $g(y, x_1, \dots, x_n) = \min_{t \leq y} P(t, x_1, \dots, x_n)$.

Teorema (la minimalizzazione limitata è ricorsiva primitiva): Se $P(t, x_1, \dots, x_n)$ appartiene ad una classe PRC C, allora $\min_{t \leq y} P(t, x_1, \dots, x_n)$ appartiene alla stessa classe PRC C.

Dimostrazione: Analizziamo la produttoria interna: $\prod_{t=0}^{u} \alpha\big(P(t, x_1, \dots, x_n)\big) = \begin{cases} 1 \; se \; y < t_0 \\ 0 \; altrimenti \end{cases}$; di questa produttoria ora ne facciamo la sommatoria, ottenendo il numero di valori strettamente minori di t_0. Quindi $g(y, x_1, \dots, x_n) = \sum_{u=0}^{y} \big(\prod_{t=0}^{u} \alpha\big(P(t, x_1, \dots, x_n)\big) \big) = t_0$. Il risultato della minimalizzazione limitata sarà $\min_{t \leq y} P(t, x_1, \dots, x_n) = \begin{cases} g(y, x_1, \dots, x_n) \; se \; (\exists t = y): P(t, x_1, \dots, x_n) \\ 0 \qquad\qquad\qquad altrimenti \end{cases}$.
Tale funzione $(n+1)$-aria è ancora in C, poiché è definita per casi e la distinzione tra i casi è data da un predicato in C (quantificazione esistenziale limitata su P), e la funzione g che appartiene anch'essa a C perché ottenuta da P applicando α e gli operatori di produttoria e sommatoria.

Vediamo come questo operatore ci permetta di fornire altri esempi di funzioni ricorsive primitive:

- Quoziente della divisione intera $\lfloor x/y \rfloor = \min_{t \leq x} \big((t+1)y > x \big)$ (con qualche vertenza poiché se $x = 0$ la funzione non dovrebbe essere definita ma per convenzione, poiché dobbiamo definire funzioni totali, poniamo che $\frac{0}{y} = 0$) è ricorsiva primitiva poiché ottenuta per minimalizzazione limitata da un predicato ricorsivo primitivo.
- Resto: $x \bmod y = x \dot{-} (y \cdot \lfloor x/y \rfloor)$

Questi esempi ci serviranno per codificare ogni S programma tramite un numero, in particolare avremmo bisogno dei numeri primi; i quali tramite la minimalizzazione possiamo dire che l'ennesimo numero primo è una funzione ricorsiva primitiva.

Funzione enumeratore di primi

Definiamo $p_0 = 0$ (per avere una funzione totale poniamo 0 come numero primo) e per $n \geq 1$ sia p_n l'ennesimo numero primo. Allora p_n è funzione ricorsiva primitiva di n.
Osserviamo, infatti che $p_{n+1} = \min_{t \leq p_n! + 1} \big(Prime(t) \wedge (t > p_n) \big)$: tra i numeri maggiori di p_n e minori o uguali a $p_n! + 1$ c'è almeno un numero primo ed è chiaro che il più piccolo primo maggiore di p_n è p_{n+1} per definizione; il numero primo esiste sempre perché $p_n! + 1$ dà sempre resto 1 se diviso per $p_1, \dots, p_n$, quindi non è multiplo di nessuno di essi. Dunque, o è primo o comunque è multiplo di un primo maggiore di p_n.
Poiché abbiamo intrecciato la ricorsione primitiva con la minimilizzazione primitiva, questa non basta a definire p_n come ricorsivo primitivo, per verificare che p_n è davvero ricorsivo primitivo, definiamo una funzione $h(y, z) = \min_{t \leq z} \big(Prime(t) \wedge (t > y) \big)$ che dà il minimo primo compreso tra $y + 1$ e z se esiste (altrimenti il predicato di minimilizzazione darà zero), ed è ricorsiva primitiva perché ottenuta tramite minimalizzazione limitata di predicato ricorsivo primitivo.
Altra definizione necessaria: Sia $R(x) = h(x, x! + 1)$, anch'essa ricorsiva primitiva.
Allora $p_0 = 0$ e $p_{n+1} = R(p_n)$ il che dimostra per ricorsione primitiva che p_n è ricorsiva primitiva.

Minimalizzazione non limitata

Dato un predicato $(n+1)$-ario $P(y, x_1, \dots, x_n)$ definiamo $\min_y P(y, x_1, \dots, x_n)$ come il minimo valore di y, se esiste, per cui $P(y, x_1, \dots, x_n)$, non definita altrimenti (quindi è funzione parziale), in simboli:
$\min_y P(y, x_1, \dots, x_n) = \begin{cases} \min\{y \in N | P(y, x_1, \dots, x_n)\} \ se \ \exists y \mid P(y, x_1, \dots, x_n) \\ \uparrow \ altrimenti \end{cases}$.

In generale questa funzione è non totale, quindi non possiamo aspettarci che sia ricorsiva primitiva anche se il predicato di partenza è ricorsivo primitivo, ma la totalità non è l'unico motivo; infatti, ci sono casi in cui non è ricorsivo primitivo anche se P lo è e anche quando la funzione risultante è totale.

Proposizione (la minimalizzazione non limitata è parzialmente calcolabile): Se $P(y, x_1, \dots, x_n)$è un predicato calcolabile, allora $\min_y P(y, x_1, \dots, x_n)$ è parzialmente calcolabile.

Dimostrazione: Il programma
$[A]$ $\boldsymbol{IF\ P(Y, X_1, \dots, X_n)\ GOTO\ E}$
$\boldsymbol{Y \leftarrow Y + 1}$
$\boldsymbol{GOTO\ A}$
calcola proprio $\min_y P(y, x_1, \dots, x_n)$ come funzione n-aria.

Funzione di Pairing ("Angoletto")

$\langle x, y \rangle = 2^x(2y+1) \dot{-} 1 = 2^x(2y+1) - 1$. Questa come funzione binaria, oltre ad essere ricorsiva primitiva, è una funzione biettiva. Quindi per ogni z esiste un'unica coppia di numeri x, y tale che z è angoletto di x, y. In simboli: $\forall \boldsymbol{z} \in \mathbb{N}, \exists! \ (\boldsymbol{x}, \boldsymbol{y}) \in \mathbb{N} \times \mathbb{N} : \boldsymbol{z} = \langle \boldsymbol{x}, \boldsymbol{y} \rangle$ (proprietà molto importante poiché ci permette di passare da coppie di numeri a numeri e viceversa).

Infatti, $2 = \langle x, y \rangle \Leftrightarrow z + 1 = 2^x(2y+1)$ e ogni intero positivo $z+1$ si scrive in modo univoco come prodotto di una potenza di 2 e un numero dispari. Questo dimostra che x e y sono univocamente determinati a partire da z (biettività), possiamo considerare dunque la sua funzione inversa, o meglio le sue due proiezioni (poiché noi lavoriamo sull'insieme N) $l(z)$ e $r(z)$ che sono date da:

$$l(z) = \min_{x \le z}(\exists y \le z \mid \langle x, y \rangle = z) \qquad r(z) = \min_{y \le z}(\exists x \le z \mid \langle x, y \rangle = z)$$

Riassumendo, la funzione angoletto è una funzione biettiva di $\mathbb{N} \times \mathbb{N}$ in $\mathbb{N}$; ricorsiva primitiva e con "inverse" anch'esse ricorsive primitive.
Ad esempio: $\langle 1,1 \rangle = 2^1(2 \cdot 1 + 1) - 1 = 5$. Mentre 23 si ottiene come $\langle x, y \rangle$ per $23 + 1 = 2^x(2y+1) \Leftrightarrow 24 = 2^3(2 \cdot 1 + 1)$.

Numero di Gödel di una $\boldsymbol{n}$-pla

$$[a_1, \dots, a_n] = \prod_{i=1}^{n} p_i^{a_i}$$

Dal teorema fondamentale dell'aritmetica, ogni intero maggiore di zero si scrive in unico modo (non si tiene conto dell'ordine) come prodotto di potenze di numeri primi.
Quindi se $z > 0$, sicuramente $\exists n \ge 1$ e una n-upla $(a_1, \dots, a_n)$ tale che $z = [a_1, \dots, a_n]$.

Per ogni n fissato il numero di Gödel definisce una funzione ricorsiva primitiva poiché definita con operatore produttoria su potenza di una funzione anch'essa ricorsiva primitiva.

Se $[a_1, \dots, a_n] = [b_1, \dots, b_m]$ con $n \le m$, allora: $\forall i \le n, a_i = b_i$ e $b_i = 0$ per $i > n$. In particolare, 1 è considerabile come numero di Gödel di una sequenza vuota (0-pla) oppure come numero di Gödel di una qualunque n-upla fatta tutta di 0.

Definiamo delle proiezioni delle inverse anche per i numeri di Gödel, nonostante non sia propriamente biettiva: $\forall \boldsymbol{i}, (\boldsymbol{x})_{\boldsymbol{i}} = \min_{\mathbf{t \le x}} \left(\neg \left(\boldsymbol{p}_{\boldsymbol{i}}^{\boldsymbol{t+1}} | \boldsymbol{x} \right) \right)$; in altre parole questa funzione ci dà l'esponente della massima potenza dell' i-esimo numero primo che divide i. Ad esempio $(24)_1 = 3$.

Per definizione di minimalizzazione limitata si ha che $(0)_i = 0$ per ogni i. Inoltre, anche $(1)_i = 0$ per ogni i ma per motivo diverso dalla minimalizzazione (semplicemente $x^0 = 1$ per ogni x).

Esercizi (con soluzioni)

1) Sia $\pi(x)$ una funzione che calcola i numeri primi minori o uguali a x. Mostrare che $\pi(x)$ è ricorsiva primitiva.
2) Sia $h(x)$ una funzione che calcola l'intero n tale che $n \le \sqrt{2}x < n + 1$. Mostrare che la funzione $h(x)$ sia ricorsiva primitiva.
3) Sia $R(x,t)$ un predicato ricorsivo primitivo e sia $g(x,y) = \max_{t \le y} R(x,t)$, ad esempio $g(x,y)$ è il più grande valore di $t \le y$ per cui $R(x,t)$ è vero; se non esiste, $g(x,y) = 0$. Provare che $g(x,y)$ è ricorsiva primitiva.
4) Sia $F(0) = 0, F(1) = 1, F(n+2) = F(n+1) + F(n)$ ($F(n)$ è l'ennesimo numero di Fibonacci). Provare che $F(n)$ è ricorsiva primitiva.

1) Il predicato $Prime(x)$ è ricorsivo primitivo, ed essendo $\pi(x) = \sum_{t=0}^{x} Prime(t)$ anch'esso è ricorsivo primitivo.
2) Esercizio simile a $|x - y|$ perché non è "veramente" una composizione: $h(x) = \lfloor \sqrt{2}x \rfloor$ (base di $\sqrt{2}x$). Abbiamo che $n \le \sqrt{2} \Leftrightarrow n^2 \le 2x^2$, mentre, $\sqrt{2}x < n + 1 \Leftrightarrow 2x^2 < (n+1)^2$, scriviamo quindi $h(x) = \min_{n \le 2x}(2x^2 < (n+1)^2)$ che è del tipo $g(y,x) = \min_{n \le y} P(n,x)$, in generale, se P è ricorsivo primitivo allora anche g lo è. In particolare, se $P(n,x) = (2x^2 < (n+1)^2)$, allora $h(x) = g(2x, x)$.
3) Basta osservare che se t_0 è il massimo valore minore o uguale a y tale che $R(x, t_0)$, allora $s = y \dot{-} t_0$ è il minore valore minore o uguale a y tale che $R(x, y \dot{-} s)$ è vero. Quindi se $\exists t \le y | R(x,t)$ avremo che $s = \min_{s' \le y} R(x, y \dot{-} s')$, ma allora $t_0 = g(x,y) = y \dot{-} \min_{s \le y} R(x, y \dot{-} s)$.
   In generale, $g(x,y) = \begin{cases} y \dot{-} \min_{s \le y} R(x, y \dot{-} s) \ \ se \ \exists t \le y \mid R(x,t) \\ 0 \qquad\qquad\qquad\qquad\quad altrimenti \end{cases}$
4) Definiamo $F'(n) = \langle F(n), F(n+1) \rangle$: si ha $F'(0) = \langle 0,1 \rangle = 2^0(2 \cdot 1 + 1) - 1 = 2$ (caso base) e $F'(n+1) = \langle F(n+1), F(n+2) \rangle = \langle F(n+1), F(n) + F(n+1) \rangle = \langle r(F'(n)), l(F'(n)) + r(F'(n)) \rangle$
   Dunque F' è ricorsiva primitiva, e poiché $\forall n, F(n) = l(F'(n))$, anche F lo è

Funzioni parzialmente calcolabili e lunghezza di un programma

Teorema: una funzione è parzialmente calcolabile se e solo se si può ottenere dalle funzioni iniziali attraverso un numero finito di applicazioni degli operatori di composizione, ricorsione primitiva e minimalizzazione non limitata.

Osservazione: grazie alla minimalizzazione non limitata non abbiamo più il vincolo di funzioni totali.

Prima di dare la definizione di lunghezza del programma ricordiamo che abbiamo già definito:

- Funzione numero di Gödel: $[x_1, \dots, x_n] = \prod_{i=1}^{n} p_i^{x_i}$ (ricorsiva primitiva)
- Funzione proiezione: $(x)_i = \min_{t \le x} \left(\neg \left(p_i^{t+1} | x \right) \right)$
- Funzione lunghezza: $Lt(x) = \min_{i \le x} \left((x)_i \neq 0 \wedge \left(\forall j \le x \left(j \le i \vee (x)_j = 0 \right) \right) \right)$
  Esprime quanti numeri primi dividono x, quindi quanto grande deve essere n per definire la minima n-upla dei divisori di x (visto che potrei aggiungere zeri per aumentare la lunghezza della n-upla)

Esempio: 40 in fattori primi è $2^3 \cdot 5$ e quindi come numero di Gödel è $[3,0,1] = [3,0,1,0,0]$; dunque avremo che $(40)_3 = 1$ e $lt(40) = 3$

Così $\forall a_1, \dots, a_n, ([a_1, \dots, a_n])_i = \begin{cases} a_i \; se \; i \leq n \\ 0 \; altrimenti \end{cases}$ e se $n \geq Lt(x)$, allora $[(x)_1, \dots, (x)_n] = x$.
Tutto questo viene utilizzato per codificare espressioni e S programmi in numeri.

Codifica di istruzioni e programmi

Enumeriamo le variabili: $\begin{matrix} Y & X_1 & Z_1 & X_2 & Z_2 & \dots \\ 0 & 1 & 2 & 3 & 4 & \dots \end{matrix}$ e, similemnte, le etichette (partiamo da 1 invece che da 0 così da assegnare il valore 0 all'istruzione senza etichetta): $\begin{matrix} A_1 & B_1 & C_1 & D_1 & E_1 & \dots \\ 1 & 2 & 3 & 4 & 5 & \dots \end{matrix}$

A ogni istruzione del linguaggio S corrisponderà dunque un certo numero.
Se I è un istruzione, definiamo il numero di I, in simboli $\#(\boldsymbol{I}) = \langle \boldsymbol{a}, \langle \boldsymbol{b}, \boldsymbol{c} \rangle \rangle$ (due volte l'angoletto) dove a è il numero dell'etichetta dell'istruzione (zero se non è etichettata), c è il numero della variabile che compare in I. Mentre b avrà uno dei seguenti valori:

- 0 se I è pigra ($V \leftarrow V$)
- 1 se è di incremento ($V \leftarrow V + 1$)
- 2 se è di decremento ($V \leftarrow V - 1$)
- $n + 2$ se è un salto all'n-sima etichetta (quindi prende anche il valore della etichetta che corrisponde al valore n)

Quindi ad ogni istruzione corrisponde un numero naturale e ad ogni numero naturale corrisponde una diversa istruzione.

Forniamo di seguito alcuni esempi:

1) Sia I l'istruzione $\boldsymbol{X \leftarrow X - 1}$ allora $\#(I) = \langle 0, \langle 2,1 \rangle \rangle = \langle 0, 2^2(2 \cdot 1 + 1) - 1 \rangle = \langle 0,11 \rangle = 2^0(2 \cdot 11 + 1) - 1 = 22$, quindi è la 22-esima istruzione.

2) Vediamo ora chi è l'istruzione J tale che $\#(J) = 37$: abbiamo che $37 = 2^a(2\langle b, c \rangle + 1) - 1 \Leftrightarrow 38 = 2^a(2\langle b, c \rangle + 1) \Rightarrow a = 1, \langle b, c \rangle = \frac{19-1}{2} = 9$; continuando risulta $9 = 2^b(2c + 1) - 1 \Rightarrow 10 = 2^b(2c + 1) \Rightarrow b = 1, c = 2$. Quindi J è l'istruzione $[\boldsymbol{A}]\; \boldsymbol{Z \leftarrow Z + 1}$.

La funzione numero di Gödel serve ad associare anche ad ogni programma un valore diverso:
Se $\mathcal{P}$ è il programma $(I_1, I_2, \dots, I_n)$, definiamo $\#(\mathcal{P}) = [I_1, \dots, I_n] - 1$ (sottraiamo 1 per dare una suriettività di quesa applicazione, affinché ogni numero naturale corrisponda ad uno ed un solo programma).

Aggiungiamo la seguente restrizione per dare anche iniettività:
Per rimediare alla non-iniettività dei numeri di Gödel, stabiliamo che un S programma valido non possa finire con l'istruzione numero 0, cioè con $Y \leftarrow Y$.
N:B: deve essere non etichettata, quella etichettata è evidentemente diversa da 0 come valore.

Esempio: Il programma numero 11 si ottiene come segue: $12 = [2,1]$:
$2 = \langle a, \langle b, c \rangle \rangle \Rightarrow a = 0, \langle b, c \rangle \Rightarrow b = 1, c = 0$ $\qquad \boldsymbol{Y \leftarrow Y + 1}$
$1 = \langle a', \langle b', c' \rangle \rangle \Rightarrow a' = 1, b' = 0, c' = 0$ $\qquad [\boldsymbol{A}]\; \boldsymbol{Y \leftarrow Y}$

Con un particolare accorgimento (vietare che l'ultima istruzione del programma sia pigra) effettivamente la corrispondenza tra numeri naturali e programmi diventa biunivoca.

Esempio di calcolo del numero di un programma. Sia $\mathcal{P}$ il programma

$$[A]\ \boldsymbol{X \leftarrow X - 1}$$
$$\boldsymbol{Y \leftarrow Y + 1}$$
$$\boldsymbol{IF\ X \neq 0\ GOTO\ A}$$

Allora $\#(\mathcal{P}) = [\#(I_1), \#(I_2), \#(I_3)] - 1$:

- $\#(I_1) = \langle a_1, \langle b_1, c_1 \rangle \rangle$ con $a_1 = 1$ perché $[A]$, $b_1 = 2$ perché decremento e $c_1 = 1$ perché X_1 e quindi $\langle 1, \langle 2,1 \rangle \rangle = \langle 1, 2^2(2 \cdot 1 + 1) - 1 \rangle = \langle 1,11 \rangle = 2^1(2 \cdot 11 + 1) - 1 = 45$
- $\#(I_2) = \langle a_2, \langle b_2 . c_2 \rangle \rangle$ con $a_2 = 0, b_2 = 1$ e $c_2 = 0$, dunque $\langle 0, \langle 1,0 \rangle \rangle = \langle 0,1 \rangle = 2$
- $\#(I_3) = \langle a_3, \langle b_3, c_3 \rangle \rangle = \langle 0, \langle 3,1 \rangle \rangle = \langle 0, 2^3(2 \cdot 1 + 1) - 1 \rangle = \langle 0,23 \rangle = 46$

Quindi $\#(\mathcal{P}) = [45,2,46] - 1 = 2^{45} \cdot 3^2 \cdot 5^{46} - 1$ (non ci si deve stupire che il numero così grande sia assegnato ad un programma così semplice, poiché ad ogni programma è assegnato un singolo numero).

Esercizi:

1) Calcola il numero dei due seguenti programmi:

	IF $X \neq 0$ GOTO A	$[B]$	IF $X \neq 0$ GOTO A
$[A]$	$X \leftarrow X + 1$		$Z \leftarrow Z + 1$
	IF $X \neq 0$ GOTO A		IF $Z \neq 0$ GOTO B
$[A]$	$Y \leftarrow Y + 1$	$[A]$	$X \leftarrow X$

2) Scrivere il programma $\mathcal{P}$ tale che $\#(\mathcal{P}) = 575$ (aiuto: $576 = 2^6 \cdot 3^2$)

8. Tesi di Church–Turing e funzioni non parzialmente calcolabili

L'esistenza di funzioni che non sono parzialmente calcolabili

Teorema: Esistono funzioni non parzialmente calcolabili.

Dimostrazione: Consideriamo per semplicità le funzioni unarie; è possibile ordinare in una successione tutte le funzioni parzialmente calcolabili in corrispondenza del numero dei programmi che li calcolano (ogni funzione ha un programma che la calcola ed ogni programma ha un suo numero, quindi è possibile ordinarle). Sia $\mathcal{P}_{(n)}$ il programma tale che $\#(\mathcal{P}_{(n)}) = n$, per ogni n. Allora tale successione è $\Psi^{(1)}_{\mathcal{P}_{(0)}}, \Psi^{(1)}_{\mathcal{P}_{(1)}}, \ldots$; a partire da questa successione costruiamo una tabella infinita dove ogni riga i è composta dai valori della funzione calcolata da $\Psi^{(1)}_{\mathcal{P}_{(i)}}(k)$ (quindi $\Psi^{(1)}_{\mathcal{P}_{(i)}}$ calcolata in k) per ogni $k \in \mathbb{N}$:

$$\begin{matrix} \Psi^{(1)}_{\mathcal{P}_{(0)}}(0) & \Psi^{(1)}_{\mathcal{P}_{(0)}}(1) & \ldots \\ \Psi^{(1)}_{\mathcal{P}_{(1)}}(0) & \Psi^{(1)}_{\mathcal{P}_{(1)}}(1) & \ldots \\ \vdots & \vdots & \ddots \end{matrix}$$

In generale l'$(n+1)$-sima riga sarà $\Psi^{(1)}_{\mathcal{P}_{(n)}}(0)\ \Psi^{(1)}_{\mathcal{P}_{(n)}}(1)\ \Psi^{(1)}_{\mathcal{P}_{(n)}}(2)\ \ldots$; dove se $\Psi^{(1)}_{\mathcal{P}_{(n)}}(k)$ non è definito, scriviamo -1 nella casella corrispondente. Chiamiamo $T_{n,m}$ il contenuto della casella (n,m) della tabella, cioè $\Psi^{(1)}_{\mathcal{P}_{(n)}}(m)$ se $\Psi^{(1)}_{\mathcal{P}_{(n)}}(m) \downarrow$, -1 altrmenti e definiamo $f(n) = T_{n,n} + 1$. Allora$f$ è totale ma non può comparire nella tabella, poiché, rispetto a qualunque riga della tabella ha almeno un valore distinto. Infatti $\forall m, f(m) = T_{m,m} + 1 = \begin{cases} \Psi^{(1)}_{\mathcal{P}(m)}(m) + 1 \text{ se } \Psi^{(1)}_{\mathcal{P}_m}(m) \downarrow \\ 0 \qquad\qquad \text{altrimenti} \end{cases}$. Quindi questa funzione f non è calcolabile, poiché se lo fosse dovrebbe coincidere con una delle righe della tabella, ma come abbiamo appena visto ciò è assurdo (tutti i valori della funzione sono diversi da almeno uno dei valori che compaiono nella tabella).

Tesi di Church-Turing

Non è possibile dimostrarla ma è intuitivamente vera; questa tesi si basa sull'osservazione che qualunque algoritmo a noi noto è possibile emularlo con un programma S (per quanto complicato).

Tesi: Ogni procedura algoritmica può essere emulata da un S programma (o macchina di turing, o formalismo equivalene), cioè: ogni funzione i cui valori possono essere calcolati algoritmicamente è (parzialmente) calcolabile (quindi esiste per essa un S programma che la calcola).

In base alla tesi di Church-Turing: Non esiste un algoritmo che, dato un programma ed un suo input, possa determinare se il programma terminerà o meno con quel dato input (vedi il predicato $HALT$).

9. Funzioni non calcolabili: argomenti per diagonalizzazione, problema della fermata

Problema della fermata

$HALT(x,y) \Leftrightarrow$ il programma numero y termina su input $x \Leftrightarrow \Psi_{\mathcal{P}(y)}(x)\downarrow$
(Il programma di numero y quando passato input x termina, quindi non va in loop).

Vedremo che $HALT$ non è calcolabile, quindi in generale non è possibile scrivere una procedura terminante che dice se un dato programma termina su un dato input.

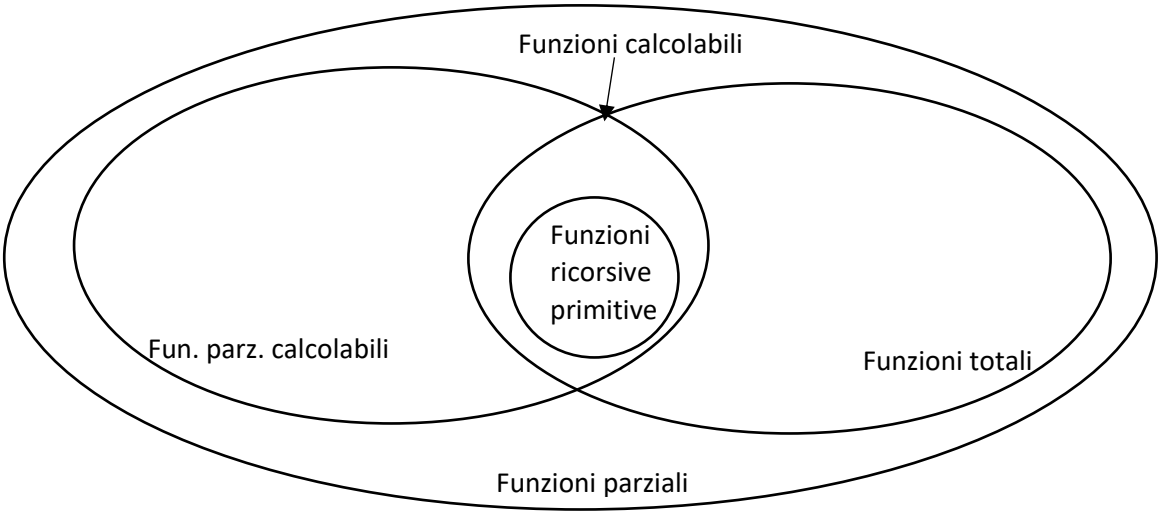
Teorema: Il predicato $HALT$ non è calcolabile.
Dimostrazione: per assurdo, sia $HALT$ calcolabile. Allora sarebbe valido il seguente programma $\mathcal{P}$:
$[A]$ $\boldsymbol{IF\ HALT(X,X)\ GOTO\ A}$. Sia $\#(\mathcal{P}) = y$. Si ha $\Psi_{\mathcal{P}}^{(1)}(x) = \begin{cases} 0\ se\ \neg HALT(x,x) \\ \uparrow\ altrimenti \end{cases}$ e quindi
$\forall x, HALT(x,y_0) \Leftrightarrow \neg HALT(x,x)$. Ma scegliendo $x = y_0$ arriviamo all'assurdo poiché $HALT(y_0,y_0) \Leftrightarrow \neg HALT(y_0,y_0)$ come volevasi dimostrare.

Congettura di Goldbach

Ogni numero pari maggiore di due è somma di due numeri primi (nessuno è riuscito a dimostrarlo).

Questo vuole dire che se scrivessimo un programma che cerca il più piccolo numero pari che non si può scrivere come somma di due primi non potremmo dire se questo programma si fermerebbe mai. Ad oggi non è noto se ciò è vero o falso, pertanto non sappiamo se avrà mai termine il programma che calcoli il $\min_{n}\Big(\neg\Big((\exists x \le 2n+4)(\exists y \le 2n+4)\big(Prime(x) \wedge Prime(y) \wedge (2n+4 = x+y)\big)\Big)\Big)$.

Schema delle funzioni da $\mathbb{N}^n$ in $\mathbb{N}$



10. Universalità; parziale calcolabilità delle funzioni universali

Funzione universale

Definiamo $\Phi^{(n)}(x_1, \dots, x_n, y) = \Psi^{(n)}_{\mathcal{P}_{(y)}}(x_1, \dots, x_n)$, intendendo in particolare che $\Phi^{(n)}(x_1, \dots, x_n, y) \downarrow \Leftrightarrow \Psi^{(n)}_{\mathcal{P}_{(y)}}(x_1, \dots, x_n) \downarrow$ (a volte la si trova anche con la notazione $\Phi^{(n)}_y(x_1, \dots, x_n)$).

Teorema (Universalità): $\forall n$, la funzione $\Phi^{(n)}$ è parzialmente calcolabile.
Dimostrazione: la funzione universale è calcolata dal programma universale (vedi dopo).

Nel compito la versione più frequente è quella a due argomenti, ovvero $\Phi^{(1)}(x, y) = \Phi(x, y)$.

Codifica degli stati di un programma

Ricordiamo che uno stato di un programma $\mathcal{P}$ è una lista di ugualgianze del tipo $V = n$, che contenga esattamente un'uguaglianza per ogni variabile V che occorre in $\mathcal{P}$ (vedi Sintassi formale del linguaggio S).

Rappresenteremo gli stati come segue: Sia V_n la variabile numero n ($Y = V_0, X_1 = V_1, Z_1 = V_2, \dots$). Allora allo stato $\{V_i = n_i | i \in I\}$ corrisponderà il numero $[n_0, n_1, \dots]$ dove si intende $n_j = 0$ se $j \notin I$.

Ad esempio, lo stato $\{Y = 0, X_1 = 2, X_2 = 3\}$ avrà il numero $[0,2,0,3]$ (abbiamo come terza variabile zero poiché Z_1 non occorre nel programma, si potrebbero mettere anche alla fine degli zeri ma sappiamo che non influisce sul numero di Gödel). Questa codifica degli stati non è né suriettiva né iniettiva.

Il programma universale $\mathcal{U}^{(n)}$

Il programma universale $\mathcal{U}^{(n)}$ estrae dall'ultima variabile di input il numero di programma da emulare; a partire da esso sarà in grado di determinare il successore di qualunque sua istantanea non terminale.

Nello scrivere il programma universale, al fine di dare una maggiore leggibilità al programma useremo anche variabili ed etichette diverse da quelle ammesse nel linguaggio S.

$Z \leftarrow X_{n+1} + 1$	$(Z = [\#(I_1), \dots, \#(I_m)]\ con\ m = Lt(Z))$
$K \leftarrow 1$	
$S \leftarrow \prod_{i=1}^{n} (p_{2i})^{X_i}$	(ricordiamo che $[a_i, \dots, a_h] = \prod_{i=1}^{h} (p_i)^{a_i}$)
$[C]$ $IF\ (K = Lt(Z) + 1 \vee K = 0)\ GOTO\ F$	(cond.: l'istantanea corrente è terminale?)
$U \leftarrow r((Z)_K)$	($Z = \langle a, \langle b, c \rangle \rangle \Rightarrow U = \langle b, c \rangle$ della k istruzione)
$IF\ l(U) = 0\ GOTO\ N$	(se $b = 0$ istruzione prigra: non fare nulla)
$P \leftarrow P_{r(U)+1}$	(numero primo corrispondente alla variabile c)
$IF\ l(U) = 1$ GOTO A	
$IF\ \neg(P \mid S)\ GOTO\ N$	($c = 0$ se il numero primo è divisore di S)
$IF\ l(U) = 2\ GOTO\ M$	
$K \leftarrow \min_{i \le Lt(Z)} (l(U) = l((Z)_i) + 2)$	(caso $b > 2$ e $P \mid S$: la minimalizzazione limitata della lunghezza del programma ritornerà la posizione della prima riga etichettata con l'etichetta del salto. Se non la trova sarà $K = 0$)
$GOTO\ C$	(aggiornato K si ricontrolla)
$[A]$ $S \leftarrow S \cdot P$	(aggiungo 1 all'esponente del primo P in S)
$GOTO\ N$	
$[M]$ $S \leftarrow \lfloor S/P \rfloor$	(decremento l'esponente, non è necessario il $GOTO\ N$ essendo l'istruzione successiva)
$[N]$ $K \leftarrow K + 1$	(Incrementiamo il contatore del programma)
$GOTO\ C$	(ripetiamo il loop)
$[F]$ $Y \leftarrow (S)_1$	(output: il massimo esponente di 2 che divide S)

Notazione per la funzione universale Φ e predicato STP

Definiamo il predicato (legato alla funzione universale) $STP^{(n)}(x_1, \dots, x_n, y, t)$ dove y è il numero del programma e t una variabile che indica il numero di passi. Il predicato sarà vero se e solo se il programma numero y $\left(\mathcal{P}_{(y)}\right)$ termina dopo al più t passi, dato l'input $(x_1, \dots, x_n)$.

Questo predicato è calcolabile (lo si potrebbe dimostrare semplicemente aggiungendo un contatore al programma della funzione universale visto precedentemente). $\forall n \geq 1, STP^{(n)}$ è ricorsivo primitivo.

Funzione di Ackermann-Péter

La seguente funzione è un esempio di funzione calcolabile ma non ricorsiva primitiva:

$$A(x,y) = \begin{cases} y+1 & se\ x = 0 \\ A(x-1,1) & se\ y = 0 \\ A\big(x-1, A(x,y-1)\big) & altrimenti \end{cases}$$ (se $x \neq 0$, il primo caso è automaticamente escluso)

Es.: $A(1,2) = A\big(0, A(1,1)\big) = A\Big(0, A\big(0, A(1,0)\big)\Big) = A\Big(0, A\big(0, A(0,1)\big)\Big) = A\big(0, A(0,2)\big) = A(0,3) = 4$

Funzione caratteristica

Se $B \subseteq \mathbb{N}^n$ si può considerare la cosiddetta funzione caratteristica (n-aria) di B, f_b tale che

$f_b(x_1, \dots, x_n) = \begin{cases} 1\ se\ (x_1, \dots, x_n) \in B \\ 0\ altrimenti \end{cases}$ (altri non è che il predicato di appartenenza a B).

Diciamo che B è ricorsivo, o calcolabile, se f_b è calcolabile. Mentre, B si dirà ricorsivo primitivo, se la funzione di appartenenza è ricorsiva primitiva.

Proposizione: $B \subseteq \mathbb{N}^n$ è in C classe PRC se e solo se lo è anche $B' = \{[x_1, \dots, x_n] | (x_1, \dots, x_n) \in B)\}$ (quindi solo se lo è anche un suo sottoinsieme)

Dimostrazione: basta legare la funzione caratteristica di B con B'; abbiamo che $x \in B' \Leftrightarrow f_{B'}(x) = 1 \Leftrightarrow f_B((x)_1, (x)_2, \dots, (x)_n) = 1 \wedge Lt(x) \leq n \wedge x > 0$. Viceversa, $(x_1, \dots, x_n) \in B \Leftrightarrow f_B(x_1, \dots, x_n) = 1 \Leftrightarrow f_{B'}([x_1, \dots, x_n]) = 1$; come volevasi dimostrare.

Teorema: Se $A, B \subseteq \mathbb{N}^n$ sono in una classe PRC C, allora anche $A \cup B, A \cap B, \bar{A} = \mathbb{N}^n \setminus A$ e $\bar{B} = \mathbb{N}^n \setminus B$ (complemento di A e B) sono in C.

Dimostrazione: è immediata, infatti, basta osservare che $f_{A \cup B}$ è il predicato di appartenenza all'unione: $f_{A \cup B} = f_A \vee f_B$. Analogamente, $f_{A \cap B} = f_A \wedge f_B, f_{\bar{A}} = \neg f_A$ e $f_{\bar{B}} = \neg f_B$ e, banalmente, siccome tutte le classi PRC sono chiuse rispetto le operazioni booleane abbiamo concluso.

Esempi:

1) Poiché $Prime$ è ricorsivo primitivo, l'insieme dei numeri primi è ricorsivo primitivo.
2) Poiché $HALT$ non è calcolabile, l'insieme $\{(x,y) \in \mathbb{N}^2 | HALT(x,y)\}$ è non ricorsivo, così come è non ricorsivo l'insieme $\{[x,y] | HALT(x,y)\}$

11. Insiemi di numeri (naturali) ricorsivi e ricorsivamente enumerabili

Insiemi ricorsivamente enumerabili

Diremo che $B \subseteq \mathbb{N}$ è ricorsivamente enumerabile (o semplicemente r.e.) se esiste f funzione parzialmente calcolabile unaria che termini solo per input $x \in B$; dunque, tale che $B = \{x \in \mathbb{N} | f(x) \downarrow\}$.

Decidibilità: procedure di decisione e semi-decisione

In base alla tesi di Church-Turing e per quanto scritto prima, B è r.e. se esiste una procedura di **semi-decisione** per B, cioè un programma (come quello che calcola f) che termina solo per gli input in B. Se invece un dato B è ricorsivo, vuol dire che per esso esiste una procedura di **decisione**, cioè un programma (come quello che calcola f_B) che per ogni x è in grado di dire in tempo finito se appartiene a B oppure no.

La ricorsione implica la ricorsivamente enumerabilità

Teorema: Se $B \subseteq \mathbb{N}$ è ricorsivo, allora è ricorsivamente enumerabile (ric. primitivo ⇒ ricorsivo ⇒ r.e.)
Dimostrazione: Se B è ricorsivo allora f_B è calcolabile e la si può usare come macro in un S programma. Dobbiamo riuscire, quindi, a scrivere un programma che termini se e solo se gli diamo B come input. A tal scopo, usiamo f_B per scrivere un programma $\mathcal{P}$ che termini solo per gli input appartenenti a B, ovvero: $[A]$ $\boldsymbol{IF\ \neg f_B(X)\ GOTO\ A}$. Allora $\Psi_{\mathcal{P}}(X) = \begin{cases} 0\ se\ x \in B \\ \uparrow\ altrimenti \end{cases}$ come volevasi dimostrare.

Un insieme è ricorsivo se e solo se il suo complemento ed esso stesso sono r.e.

Teorema: $B \subseteq \mathbb{N}$ è ricorsivo se e solo se B e $\bar{B}$ sono entrambi ricorsivamente enumerabili.
Dimostrazione: Implicazione ⇒: per un teorema visto in precedenza (capitolo funzione caratteristica) se B è ricorsivo anche il suo complemento è ricorsivo, ne segue che, per il teorema precedente, sia B che $\bar{B}$ sono r.e.. Implicazione ⇐: se B e $\bar{B}$ sono r.e., esistono due funzioni f e g parzialmente calcolabili tali che $B = \{x \in \mathbb{N} | f(x)\downarrow\}$ e $\bar{B} = \{x \in \mathbb{N} | g(x)\downarrow\}$ (quindi esistono programmi che li calcolano). Siano rispettivamente p e q i numeri di programmi che calcolano f e g. Scriviamo, così da dimostrare l'implicazione, un programma per f_B:

$[A]$ $\boldsymbol{IF\ STP(X,p,Z)\ GOTO\ C}$
$\boldsymbol{IF\ STP(X,q,Z)\ GOTO\ E}$
$\boldsymbol{Z \leftarrow Z+1}$ (incrementiamo il numero di passi da aspettare)
$\boldsymbol{GOTO\ A}$
$[C]$ $\boldsymbol{Y \leftarrow Y+1}$

Esercizi svolti (o quasi)

1) Sia $H_1(x) = \begin{cases} 1\ se\ \Phi(x,x)\downarrow \\ \uparrow\ altrimenti \end{cases}$. Mostrare che H_1 è parzialmente calcolabile.

La funzione universale è parzialmente calcolabile e, quindi, si può richiamare come macro; dunque, una possibile soluzione dell'esercizio è il seguente programma:

$\boldsymbol{Z \leftarrow \Phi(X,X)}$
$\boldsymbol{Y \leftarrow Y+1}$

Altra possibile soluzione:

$[A]$ $\boldsymbol{IF\ STP(X,X,Z)\ GOTO\ B}$
$\boldsymbol{Z \leftarrow Z+1}$
$\boldsymbol{GOTO\ A}$
$[B]$ $\boldsymbol{Y \leftarrow 1}$

2) Dare un programma $\mathcal{P}$ tale che $H_{\mathcal{P}}(x_1,x_2)$, definito come $H_{\mathcal{P}}(x_1,x_2) \Leftrightarrow \mathcal{P}$ si ferma con input x_1,x_2, è non calcolabile.

Partire da un predicato che non sia calcolabile (esempio $HALT$) e prendere un programma $\mathcal{P}$ che fa in modo che questo programma diventi $HALT$. Tale programma è proprio il programma universale su due argomenti, il più semplice $\mathcal{U}$ basta per svolgere l'esercizio.

Chiusura rispetto l'intersezione e l'unione

Teorema: Se B, C sono insiemi di numeri ricorsivamente enumerabili anche $B \cap C$ lo è.
Dimostrazione: Per ipotesi B, C sono r.e., quindi esistono f, g parzialmente calcolabili e tali che $B = \{x \in \mathbb{N} | f(x)\downarrow\}$ e $C = \{x \in \mathbb{N} | g(x)\downarrow\}$, di conseguenza, possiamo usare queste due funzioni come macro di un programma, il quale termina solo per gli input che appartengono all'intersezione (altrimenti non termina): $\begin{matrix} \boldsymbol{Z \leftarrow f(X)} \\ \boldsymbol{Z \leftarrow g(X)} \end{matrix}$ dove $X \in B \cap C$. Questo programma praticamente funziona nel seguente modo: se $X \in f$ vado avanti, e se lo stesso X appartiene anche a g allora l'istruzione termina e restituisco 0, mentre se $X \notin B \vee X \notin C$ allora l'istruzione corrispettiva andrà in loop. C.V.D.

Teorema: Se B, C sono insiemi di numeri ricorsivamente enumerabili anche $B \cup C$ lo è.
Dimostrazione: esistono f, g parzialmente calcolabili tali che $B = \{x \in \mathbb{N} | f(x)\downarrow\}$ e $C = \{x \in \mathbb{N} | g(x)\downarrow\}$, siano, inoltre, p e q i numeri di programmi che calcolano f e g, rispettivamente. Il seguente programma termina solo per gli $X \in B \cup C$ (ovvero che termina se e solo se $X \in B \vee X \in C$):

$[A]$ $\boldsymbol{IF\ STP(X,p,Z)\ GOTO\ E}$
$\boldsymbol{IF\ STP(X,q,Z)\ GOTO\ E}$
$\boldsymbol{Z \leftarrow Z+1}$
$\boldsymbol{GOTO\ A}$

Osservazione: Generalmente per verificare l'input di due insiemi r.e. non si può fare in sequenza (come nel caso dell'intersezione) poiché potrebbe succedere che l'elemento non appartiene al primo insieme oppure non appartiene al secondo; quindi, per verificare l'appartenenza almeno ad uno dei due insiemi si usa STP.

12. Insieme diagonale e antidiagonale

Dominio dell'insieme di definizione del programma numero n

Definiamo, per $n \in \mathbb{N}$, l'insieme $W_n = \{x \in \mathbb{N} | \Phi(x,n) \downarrow\}$ (è r.e. essendo Φ funzione unaria parzialmente calcolabile). L'insieme si può scrivere anche come $W_n = \left\{x \in \mathbb{N} \middle| \Psi^{(1)}_{\mathcal{P}_{(n)}}(x) \downarrow\right\}$

Teorema di enumerazione: B è r.e. $\Leftrightarrow \exists n \in \mathbb{N} : B = W_n$

Dimostrazione: B è r.e. $\Leftrightarrow \exists f$ parzialmente calcolabile tale che $B = \{x \in \mathbb{N} | f(x) \downarrow\} \Leftrightarrow \exists n : f = \Psi^{(1)}_{\mathcal{P}_{(n)}}$ e $B = \left\{x \in \mathbb{N} \middle| \Psi^{(1)}_{\mathcal{P}_{(n)}} \downarrow\right\} \Leftrightarrow \exists n : B = W_n$. C.V.D.

Insieme diagonale e sue proprietà

Definiamo l'insieme **diagonale** $K = \{n \in \mathbb{N} | n \in W_n\} = \{n \in \mathbb{N} | \Phi(\mathrm{n},\mathrm{n}) \downarrow\}$.
N.B.: Le seguenti notazioni sono tra loro equivalenti: $n \in W_n \Leftrightarrow \Phi(\mathrm{n},\mathrm{n}) \downarrow \Leftrightarrow HALT(n,n) \Leftrightarrow n \in K$.

Proposizione 1: K è ricorsivamente enumerabile ma non ricorsivo.
Dimostrazione: K è r.e. in quanto il seguente programma termina solo per gli $x \in K$: $\boldsymbol{Z \leftarrow \Phi(X,X)}$
Sono due le possibili dimostrazioni per dire che K non è ricorsivo:

1) Dimostriamo che il suo complemento $\overline{K} = \mathbb{N} \setminus K$ (insieme **antidiagonale**) non è r.e.: se per assurdo $\overline{K}$ fosse r.e., in base al teorema di enumerazione, dovrebbe esistere i tale che $\overline{K} = W_i$. Ma allora $i \in \overline{K} \Leftrightarrow i \in W_i \Leftrightarrow i \in K$ e ciò è assurdo poiché K e $\overline{K}$ sono disgiunti.
2) La funzione caratteristica f_K verifica che $f_K(x) = \begin{cases} 1\ se\ x \in K \\ 0\ altrimenti \end{cases}$ ma $x \in K \Leftrightarrow HALT(x,x)$, e ciò significa che $f_K(x) = HALT(x,x)$ ma $HALT(x,x)$ abbiamo dimostrato essere non calcolabile e quindi K non è ricorsivo.

Proposizione 2: Se $B \subseteq \mathbb{N}$ è r.e., esiste R predicato binario e ricorsivo primitivo tale che $B = \left\{x \in \mathbb{N} \middle| \exists t\big(R(x,t)\big)\right\}$ (quindi se B è r.e. lo si può scrivere come $R(x,t)$ per un certo valore di t)
Dimostrazione: ricordando che il predicato STP è ricorsivo primitivo, se B è r.e. possiamo suppore $B = W_n$ per un certo n. Allora $B = \left\{x \in \mathbb{N} \middle| \exists t \left(STP^{(1)}(x,n,t)\right)\right\}$ (il programma numero n termina con input x in t passi) e dato che $W_n = \{x \in \mathbb{N} | HALT(x,n)\}$ la proposizione è dimostrata.

Alle precedenti notazioni equivalenti possiamo aggiungere anche $\exists t : STP(n,n,t)$, ricapitolando quindi saranno equivalenti: $n \in W_n \Leftrightarrow \Phi(\mathrm{n},\mathrm{n}) \downarrow \Leftrightarrow HALT(n.n) \Leftrightarrow n \in K \Leftrightarrow \exists t : STP(n,n,t)$.

Teorema 3: Sia $B \neq \emptyset$ e B r.e.. Allora esiste f ricorsiva primitiva tale che $B = \{f(n) | n \in \mathbb{B}\}$.
Dimostrazione: usiamo la proposizione precedente: $\exists R$ predicato ricorsivo primitivo tale che $B = \left\{x \in \mathbb{N} \middle| \exists t\big(R(x,t)\big)\right\}$. Dato che $B \neq \emptyset$ scegliamo un elmento $x_0 \in B$, e definiamo una funzione $f(x) = \begin{cases} l(x)\ se\ R\big(l(x),r(x)\big) \\ x_0 \quad altrimenti \end{cases}$. Se $n \in B, \exists t \mid R(n,t)$. Dato $x = \langle n,t \rangle$, si ha che $f(\langle n,t \rangle) = \begin{cases} l(\langle n,t \rangle) = n\ se\ R \\ x_0 \qquad altrimenti \end{cases}$
$x_0 \in B$ per supposizione, quindi resta da verificare che $\forall x \in \mathbb{N}, f(x) \in B$: infatti $f(x) = l(x)$ se $R\big(l(x),r(x)\big)$, ma dato che $B = \left\{n \in \mathbb{N} \middle| \exists t\big(R(n,t)\big)\right\}$, in tal caso si ha $l(x) \in B$ (per $n = l(x)$ esiste $t = r(x)$ tale che $R(n,t)$), come volevasi dimostrare.

Teorema 4: Sia f unaria e parzialmente calcolabile e $B = \{f(n)|f(n)\downarrow\}$. Allora B è r.e.

Dimostrazione: Sia p il numero di un programma che calcola f, scriviamo un programma che termina solo sui valori assunti dalla funzione f (quindi solo se $X = f(n)$ per qualche n):

$[A]$ $\boldsymbol{IF\ \neg STP(Z_1, p, Z_2)\ GOTO\ B}$ (se passa all'istruzione successiva significa che $Z_1 \in f$)
$\boldsymbol{Z_3 \leftarrow f(Z_1)}$ (se Z_1 è un valore su cui f non è definito
$\boldsymbol{IF\ Z_3 = X\ GOTO\ E}$ allora il salto verrà effettuato)
$[B]$ $\boldsymbol{Z_1 \leftarrow Z_1 + 1}$ (il predicato è vero solo se il programma numero p non termina
$\boldsymbol{IF\ Z_1 \leq Z_2\ GOTO\ A}$ con input Z_1 in Z_2 passi, la nostra condizione è, infatti, $\neg STP$)
$\boldsymbol{Z_2 \leftarrow Z_2 + 1}$
$\boldsymbol{Z_1 \leftarrow 0}$
$\boldsymbol{GOTO\ A}$

Infatti, se n è tale che $f(n)\downarrow$ e $X = f(n)$ allora esiste un valore t tale che $STP(n, p, t)$, e prima o poi il programma raggiungerà i valori $Z_1 = n$ e $Z_2 \geq t$. L'unica condizione di terminazione del programma è attraverso l'istruzione numero 3, cioè quando $X \in f$, ovvero si trova un tale n, C.V.D.

Teorema "riassuntivo"

Si possono riassumere i risultati precedenti nel seguente teorema:

Teorema di equivalenza: Sia $B \subseteq \mathbb{N}, B \neq \emptyset$. Sono equivalenti le seguenti affermazioni:

1) B è r.e., cioè $\exists g$ parzialmente calcolabile tale che $B = \{x \in \mathbb{N}|g(x)\downarrow\}$
2) $B = \{f(n)|n \in \mathbb{N}\}$ per qualche f ricorsiva primitiva
3) $B = \{f(n)|n \in \mathbb{N}\}$ per qualche f calcolabile
4) $B = \{f(n)|f(n)\downarrow\}$ per qualche f parzialmente calcolabile

Dimostrazione: abbiamo già dimostrato che $1 \Rightarrow 2$ e $4 \Rightarrow 1$, mentre le implicazioni restanti, $2 \Rightarrow 3 \Rightarrow 4$, sono banali ($f$ è ricorsiva primitiva $\Rightarrow$ f calcolabile $\Rightarrow$ f parzialmente calcolabile).

Esercizi:

1) Dimostrare che $\{\langle x, y\rangle | x \in W_y\}$ è r.e..
2) Sia B r.e. e $C \leq B$. È sempre vero che C è r.e.?
3) Sia $B \cup C$ r.e.. È sempre vero che B e C sono r.e.?
4) Dimostrare che non esiste una f calcolabile tale che $f(x) = \Phi(\mathrm{x}, \mathrm{x}) + 1$ se $\Phi(x, x)\downarrow$.

13. Macchine di Turing.

Rappresentazione unaria, binaria e n-aria

Incremento e decremento non sono necessariamente operazioni "più elementari" possibili, ma lo sono se usiamo una notazione in base 1, ovvero una rappresentazione unaria (scriviamo il numero n come una stringa di n "uni"; i.e.: $3 = 111$). Non dovrebbe stupire che un analogo di quello fatto per il linguaggio S si può fare con altre rappresentazioni di numeri, come quella in base 2 (binaria), o qualsiasi altra rappresentazione n-aria, dove al posto di incremento e decremento avremo altri tipi di operazioni elementari.

Praticamente si possono ottenere versioni alternative del linguaggio S, e questi linguaggi hanno la stessa potenza di calcolo del linguaggio S, si dice quindi che i linguaggi sono equivalenti.

Modelli diversi: macchina di Turing

Una macchina di Turing si può pensare come costituita da una "testa" che scorre su un "nastro" infinito (praticamente fa da memoria) sul quale può leggere e scrivere secondo delle determinate "istruzioni".

Una macchina di Turing calcola funzioni in base a tali istruzioni: partendo da uno stato iniziale ben preciso e da una rappresentazione di un valore di input sul nastro, l'output corrispondente sarà scritto sul nastro al termine del calcolo (se termina).

Formalmente, una macchina di Turing è data dalla quadrupla $(\boldsymbol{Q}, \boldsymbol{q_1}, \boldsymbol{A}, \boldsymbol{I})$ dove:

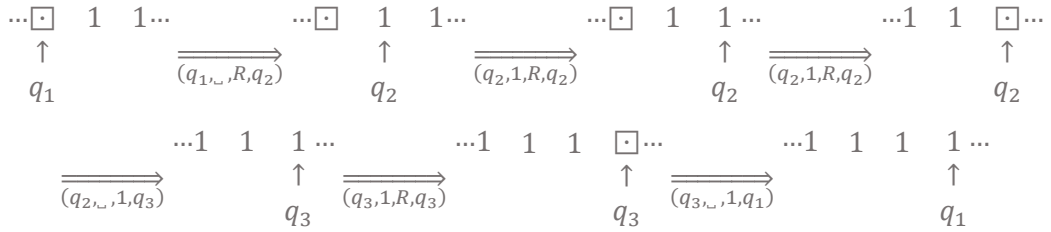
- $\boldsymbol{Q}$: insieme finito di **stati**
- $\boldsymbol{q_1} \in Q$: **stato iniziale**
- $\boldsymbol{A}$: **alfabeto** (sempre insieme finito)
- $\boldsymbol{I} \subseteq Q \times (A \cup \{\text{␣}\}) \times (A \cup \{\text{␣}, L, R\}) \times Q$: quindi l'insieme di istruzioni è formato da un qualunque insieme di quadruple della forma precedentemente descritta (␣ = spazio vuoto).

Il calcolo di una macchina di Turing si svolge come segue:

- Ad ogni istante la macchina è in uno stato $q \in Q$ ben preciso e legge un simbolo $a \in A \cup \{\text{␣}\}$ sul nastro
- Verifica se nel proprio insieme di istruzioni I ce n'è una che inizia con la coppia stato-simbolo; se la trova, la "esegue", cioè se ad esempio $(q, a, b, q') \in A$, con $b \in A \cup \{\text{␣}, L, R\}$ e $q' \in Q$ allora:
  - Sostituisce nella posizione corrente sul nastro a con b se $b \in A \cup \{\text{␣}\}$
  - Oppure, si muove a sinistra o a destra se b è L o R, rispettivamente.
  - In entrambi i casi poi la macchina passa allo stato q'

  N.B.: la coppia stato simbolo è unica, non troverai mai coppie uguale nello stesso insieme di istruzioni
- Se non trova la suddetta istruzione la macchina si ferma ed il calcolo termina (come un S programma quando non trova l'istruzione successiva).

Esempio: Sia la macchina $\mathcal{M} = (Q, q_1, A, I)$ con insieme di stati $Q = \{q_1, q_2, q_3\}$, alfabeto $A = \{1\}$ e insieme di istruzioni $I = \{(q_1, \text{␣}, R, q_2), (q_2, 1, R, q_2), (q_2, \text{␣}, 1, q_3), (q_3, 1, R, q_3), (q_3, \text{␣}, 1, q_1)\}$.
Cerchiamo di capire quale funzione rappresenti tramite un esempio di calcolo (quello alla destra del nastro rappresenta il nostro input, quindi in questo caso è 2). Ma prima diamo un esempio di come si legge una istruzione: $(q_2, 1, R, q_2) =$ se sei nello stato q_2 e leggi 1 allora spostati a destra e rimani in q_2.



e qui la macchina si ferma poiché non esistono nelle istruzioni la coppia $(q_1, 1)$. Si noti che questa macchina somma semplicemente 2 all'input (rappresentazione unaria); quindi calcola la funzione $f(x) = x + 2$

Linguaggi (insiemi di parole) r.e. e ricorsivi

L'equivalenza tra linguaggio S e macchina di Turing (stabilita dalla tesi di Church-Turing), ci dice, ad esempio, che un sottoinsieme di $\mathbb{N}$ è ricorsivamente enumerabile se e solo se è l'insieme degli input sui quali il calcolo di una $\mathcal{M}$ macchina di Turing termina, tali input si dicono **accettati** dalla macchina $\mathcal{M}$.

Tutto ciò si applica a qualsiasi alfabeto A (non devono rappresentare necessariamente le cifre di un certo sistema di numerazione): possiamo, infatti, parlare di stringhe (o parole) di input accettate o meno da una macchina di Turing, che costituiscono il **linguaggio accettato** (insieme di parole per cui la macchina termina) della macchina di Turing in questione.

L'insieme di tutte le stringhe su A, inclusa la **stringa vuota** $\boldsymbol{\varepsilon}$, si denota con il simbolo $\boldsymbol{A^*}$; un **linguaggio** è semplicemente un sottoinsieme di $\boldsymbol{A^*}$.

Ricapitolando, un linguaggio è ricorsivamente enumerabile se è accettato da una macchina di Turing.

14. Automi finiti e linguaggi regolari.

Modelli di calcolo su stringhe: DFA

Gli automi finiti deterministici, conosciuti come DFA si possono pensare come macchine di Turing che non scrivono più su nastro ma leggono lo stato (che è l'unica forma di memoria per un DFA) e passano obbligatoriamente ad un altro.

Formalmente, un DFA è una quintupla $\mathcal{M} = (Q, A, \delta, q_1, F)$ dove:

- $\boldsymbol{Q}$ è un insieme finito di **stati** di $\mathcal{M}$
- $\boldsymbol{A}$ è un insieme finito detto **alfabeto**
- $\boldsymbol{\delta}$: $Q \times A \to Q$ è la funzione di **transizione** di $\mathcal{M}$
- $\boldsymbol{q_1} \in Q$ è lo **stato iniziale**
- $\boldsymbol{F} \subseteq Q$ è l'insieme di stati **terminali** (o di **accettazione**) di $\mathcal{M}$

Linguaggio accettato e linguaggio regolare

Per un automa $\mathcal{M}$ definiamo $\boldsymbol{\delta^*}$: $Q \times A^* \to Q$ come segue: $\begin{matrix} \forall q \in Q, \delta^*(q, \varepsilon) = q \text{ (caso base)} \\ \forall u \in A^*, a \in A, \delta^*(q, ua) = \delta(\delta^*(q, u), a)' \end{matrix}$

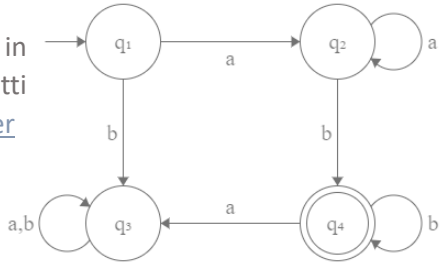
praticamente questa funzione δ^* da uno stato q ed una stringa u ritorna lo stato q dell'automa

Chiamiamo **linguaggio accettato** da $\mathcal{M}$ l'insieme $L(\mathcal{M}) = \{w \in A^* | \delta^*(q_1, w) \in F\}$ (l'insieme delle coppie stato iniziale-stringa), diremo inoltre che $L \subseteq A^*$ è **regolare** se esiste un automa finito deterministico $\mathcal{M}$ sull'alfabeto A tale che $L = L(\mathcal{M})$ (quindi è un linguaggio regolare se è accettato da un DFA).

Esempio: Sull'alfabeto $\{a, b\}$, il linguaggio $L = \{a^n b^m | n, m > 0\}$ risulta essere regolare. Infatti, è accettato dal seguente DFA: $Q = \{q_1, q_2, q_3, q_4\}, F = \{q_4\}$ (N.B.: L non contiene parole che iniziano per b).

δ	a	b
q_1	q_2	q_3
q_2	q_2	q_4
q_3	q_3	q_3
q_4	q_3	q_4

I diagrammi di transizione in questo documento sono stati fatti su Finite State Machine Designer



Si noti che dallo stato q_3 non è possibile uscire, uno stato così viene detto **pozzo**.

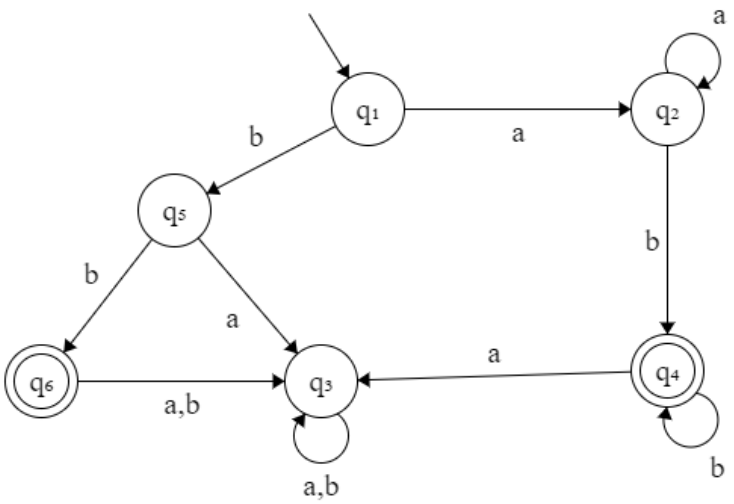
Diagramma di transizione (di stato dell'automa)

L'idea è rappresentare ogni stato dell'automa attraverso un cerchio (viene anche conosciuto come diagramma a bolla), un automa a stati finiti può essere rappresentato da un grafo orientato in cui:

- I nodi rappresentano i possibili stati del sistema;
- Gli archi rappresentano le transizioni collegate ai possibili eventi;
- Lo stato iniziale è indicato da una freccia entrante;
- Gli stati di accettazione sono doppiamente cerchiati;

Negli esercizi del compito è molto probabile che venga chiesto solo il diagramma di transizione e non la rappresentazione tabellare.

Diamo un esempio (oltre di un altro diagramma a bolla) di come lo stato terminale non è necessario che sia unico, a differenza di quello iniziale:
$Q = \{q_1, q_2, q_3, q_4\}, F = \{q_4\}, L' = L \cup \{bb\}$ (le parole dell'esempio precedente con l'aggiunta di bb), ovvero sull'alfabeto $\{a, b\}$, $L' = \{a^n b^m | n, m > 0\} \cup \{bb\}$. L'automa sarà dunque il seguente:

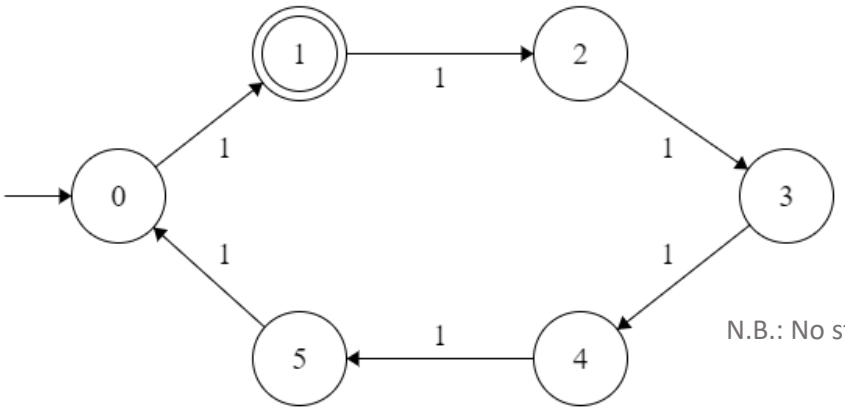


Proposizione: Sia $L \subseteq A^*$ e sia $A \subseteq A'$. Allora esiste un DFA su A che accetta L se e solo se ne esiste uno su A' (la regolarità di un linguaggio non dipende dall'alfabeto).

Dimostrazione: Sia $\mathcal{M} = (Q, A, \delta, q_1, F)$ un DFA tale che $L(\mathcal{M}) = L$. Allora per costruire un DFA $\mathcal{M}'$ su A' tale che $L(\mathcal{M}') = L$ basta aggiungere uno stato pozzo: $\mathcal{M}' = \left(Q \cup \{q_p\}, A', \delta', q_1, F\right)$, con (definizione per casi) $\delta'(q,a) = \begin{cases} \delta(q,a) \; se \; q \in Q \; e \; a \in A \\ q_p \qquad altrimenti \end{cases}$ (abbiamo così esteso l'alfabeto A ad uno più grande). Viceversa, sia $\mathcal{M}' = (Q', A', \delta', q_1', F')$ un DFA su A' che accetta L. Allora $\mathcal{M} \coloneqq \left(Q', A, \delta = \delta'_{|Q' \times A}, q_1', F'\right)$ (δ è la restrizione di δ' sul dominio $Q' \times A$) è un DFA su A, e $L(\mathcal{M}) = L$ poiché le parole di L contengono solo lettere in A (abbiamo così mostrato che da un sottoinsieme di A^* sia possibile definire un automa solo sull'alfabeto A); come volevasi dimostrare.

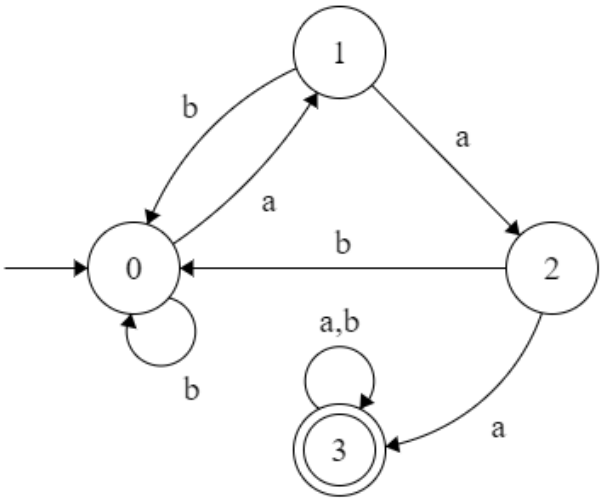
Esempi di sintesi e di analisi

Esempio di sintesi: Dimostriamo che un linguaggio $L_1 = \{1^{6k}1 | k \geq 0\}$ sia regolare. Per la proposizione precedente possiamo considerare l'alfabeto che contenga solo la stringa 1, essendo il linguaggio superiore la ripetizione di $6k + 1$ volte "1". Si noti che stiamo descrivendo nient'altro che il resto di modulo 6 delle volte che leggo uno, quindi è evidente che ci basta, anzi è necessario e sufficiente un automa di 6 stati (li numeriamo da 0 a 5 così da avere effettivamente il valore del resto per quel determinato stato)

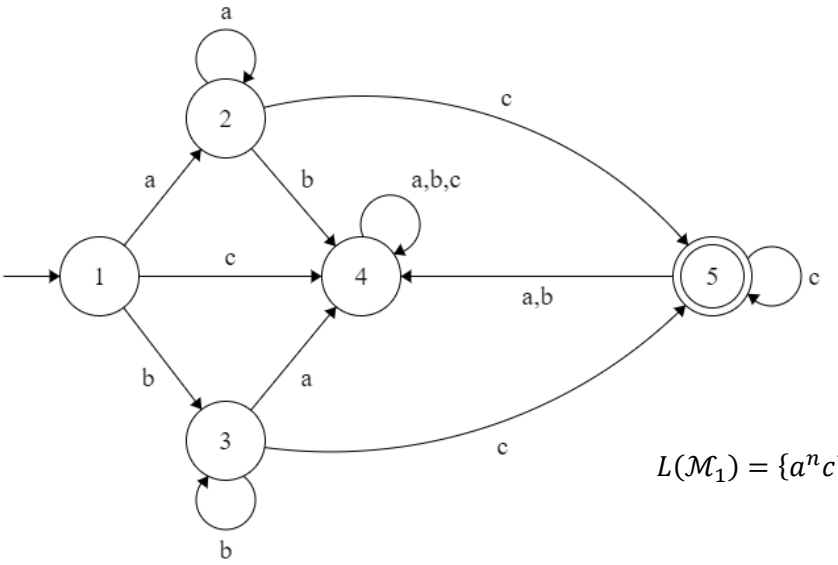


Qualunque parola del linguaggio L' può essere rappresentata aggiungendo il numero di 1 necessari

Altro esempio di sintesi: Sia $A = \{a, b\}, L_2 = \{w \in A^* | w \text{ contiene 3 a consecutive}\}$ bisogna tener traccia solo del numero di a, e quindi saranno necessari almeno 4 stati: uno di accettazione, e uno per ciascun numero di occorrenze di a alla fine della parola letta finora (numeriamo gli stati in base alle occorrenze):



Esempio di analisi (scrivere il linguaggio accettato dato un automa):



δ_1	a	b	c
q_1	q_2	q_3	q_4
q_2	q_2	q_4	q_5
q_3	q_4	q_3	q_5
q_4	q_4	q_4	q_4
q_5	q_4	q_4	q_5

$$F_1 = \{q_5\}$$

$$L(\mathcal{M}_1) = \{a^n c^m | n, m > 0\} \cup \{b^n c^m | n, m > 0\}$$

N.B.: Dato $\mathcal{M} = (Q, A, \delta, q_1, F)$ DFA, si ha che $\varepsilon \in L(\mathcal{M}) \Leftrightarrow q_1 \in F$

Automi finiti non deterministici

Un automa a stati finiti non deterministico, anche conosciuto come NDFA (o NFA) è un quintupla $\boldsymbol{\mathcal{M} = (Q, A, \delta, q_1, F)}$ con:

- Q insieme finito di **stati**
- A **alfabeto** (finito)
- $\boldsymbol{\delta}$: $\boldsymbol{Q \times A \to \mathcal{P}(Q)}$ funzione di **transizione** (è proprio qui la differenza rispetto un DFA, infatti il codominio non è più Q ma l'insieme delle sue parti)
- $q_1 \in Q$ **stato iniziale**
- $F \subseteq Q$ **stati terminali** (o stati di accettazione)

Un NDFA accetta $w \in A^*$ se leggendola una lettere alla volta, a partire dallo stato iniziale, arriva a un insieme di stati che ne contenga almeno uno terminale. Formalmente, definiamo $\delta^*: Q \times A^* \to \mathcal{P}(Q)$ in modo che $\forall q \in Q, \delta^*(q, \varepsilon) = \{q\}$ (si ricorda che porta ad un insieme di stati e non a singoli stati) e

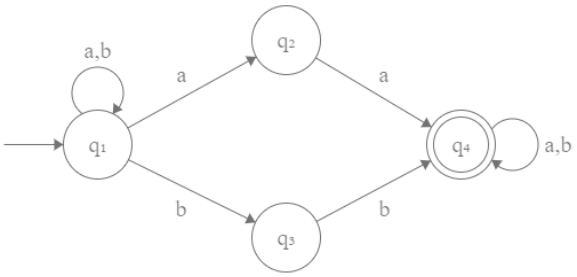
$$\forall u \in A^*, \forall a \in A, \delta^*(q, ua) = \bigcup_{q' \in \delta^*(q,u)} \delta(q', a)$$

(funzione di iterazione che ci serve per dare una dichiarazione formale del linguaggio accettato dall'automa) e allora $\boldsymbol{L(\mathcal{M}) = \{w \in A^* | \delta^*(q_1, w) \cap F \neq \emptyset\}}$ è il **linguaggio accettato**.

Proposizione: Se $L \subseteq A^*$ è regolare, esiste un NDFA $\mathcal{M}'$ tale che $L(\mathcal{M}') = L$ (possiamo rendere non deterministico un automa deterministico).
Dimostrazione: Dato un DFA $\mathcal{M} = (Q, A, \delta, q_1, F)$ che accetta L, allora $\mathcal{M}' = (Q, A, \delta', q_1, F)$, con $\delta'(q, a) = \{\delta(q, a)\}$ per ogni $q \in Q$ e $a \in A$, è un NDFA tale che $L(\mathcal{M}') = L(\mathcal{M}) = L$. CVD.

Il diagramma di transizione di un NDFA si disegna analogamente al caso deterministico, ma da ogni stato potranno uscire più archi etichettati con la stessa lettera (o anche nessun arco).

Esempio di sintesi di NDFA: Disegnare il diagramma di transizione di un NDFA che accetti tutte le parole su $\{a, b\}$ che contengano due a o due b consecutive (aa o bb): $\delta^*(q_1, ab) = \{q_1\}$



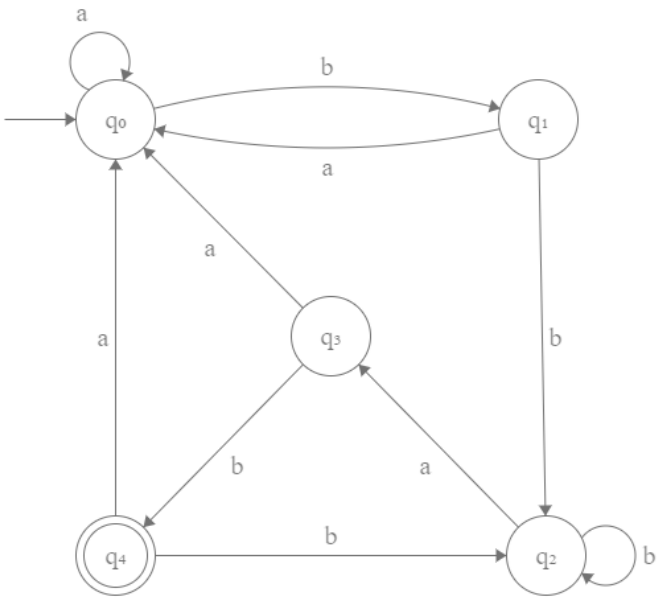
Il vantaggio di un NDFA è che possiamo considerare solo i cammini rilevanti.

Esercizi svolti da me

1) In ciascuno dei seguenti esempi, un alfabeto A e un linguaggio L sono indicati con $L \subseteq A^*$. Mostrare per ogni caso che L è regolare costruendo un automa che lo accetti.

a. $A = \{a, b\}$; L consiste in tutte le parole che terminano con la stringa $bbab$.

$$L = \{w \in A^* | w \text{ termina con } bbab\}$$



$$q_0 = \cdots a + \begin{cases} a = q_0 \\ b = q_1 \end{cases}$$

$$q_1 = \cdots b + \begin{cases} a = q_0 \\ b = q_2 \end{cases}$$

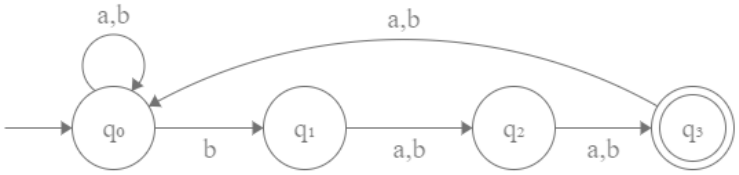
$$q_2 = \cdots bb + \begin{cases} a = q_3 \\ b = q_2 \end{cases}$$

$$q_3 = \cdots bba + \begin{cases} a = q_0 \\ b = q_4 \end{cases}$$

$$q_4 = \cdots bbab + \begin{cases} a = q_0 \\ b = q_2 \end{cases}$$

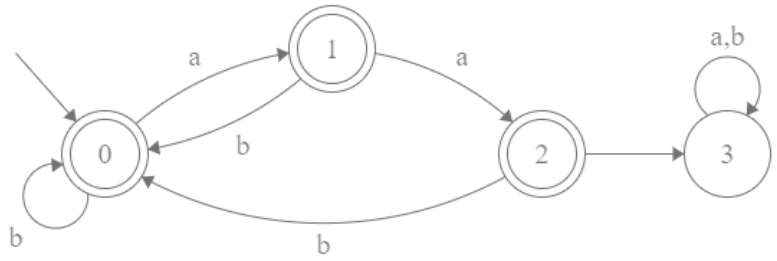
b. $A = \{a, b\}$; L consiste in tutte le stringhe $s_1 s_2, \ldots s_n$ in cui $s_{n-2} = b$ (Nota che L non contiene stringhe di lunghezza minore di 3)

$$L = \{w \in A^* | w = w' bxy \text{ con } w' \in A^* \wedge x, y \in A\}$$



c. $A = \{a, b\}$; L consiste in tutte le parole che non contengono tre a consecutive

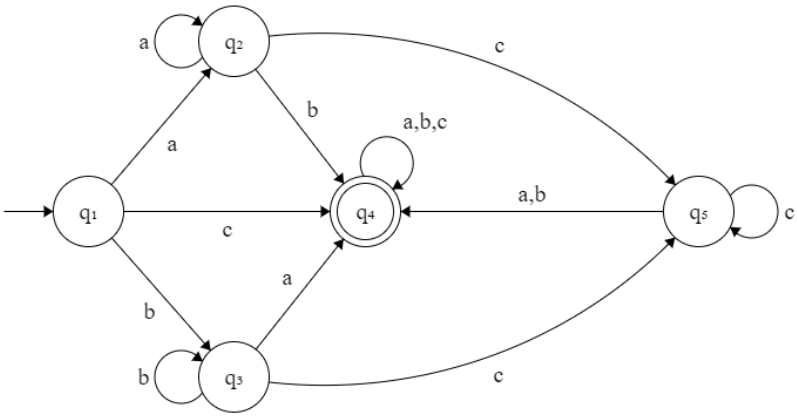
Ho dato come nomi degli stati il numero di occorrenze sequenziali di a



2) Descrivi il linguaggio accettato da ogni seguente automa. Per ogni caso lo stato iniziale è q_1.

a. $F_2 = \{q_4\}$

δ_2	a	b	c
q_1	q_2	q_3	q_4
q_2	q_2	q_4	q_5
q_3	q_4	q_3	q_5
q_4	q_4	q_4	q_4
q_5	q_4	q_4	q_5

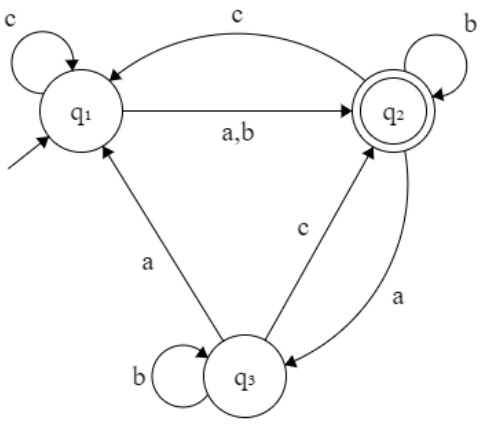


L consiste in tutte le parole che iniziano per c, oppure per una sequenza di a e poi b, oppure per una sequenza di b e poi a, o ancora per una sequenza di a (o di b) seguita da una sequenza di c e poi almeno una a o b

b. $F_3 = \{q_2\}$

δ_3	a	b	c
q_1	q_2	q_2	q_1
q_2	q_3	q_2	q_1
q_3	q_1	q_3	q_2

Corretto dal Prof.



Consideriamo prima i modi che ci sono per arrivare da q_1 a q_1 (passando una sola volta per lo stato iniziale): c oppure $\{a, b\} \cdot \{b, ab^n c\}^m \cdot \{c, ab^k a\}$, quindi il linguaggio descritto sarà: $(\{c\} \cup \{a, b\} \cdot \{b, ab^{n_1} c\}^{n_2} \cdot \{c, ab^{n_3} a\})^{n_4} \cdot \{a, b\} \cdot \{b, ab^{n_5} c\}^{n_6}$ (n volte da q_1 a q_2 e poi gli altri percorsi che non passano più per q_1)

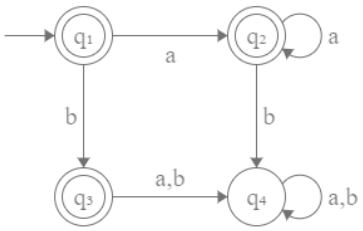
3) Sia $A = \{s_1, \dots, s_n\}$. Quanti automi finiti con esattamente $m > 0$ stati esistono su A?

$\mathcal{M} = (Q, A, \delta, q_1, F)$, l'insieme degli stati e l'alfabeto sono uguali per ogni automa; dunque, gli elementi della quintupla che possono variare sono:

- $\delta: Q \times A \to Q$: per $|Q|^{|Q \times A|} = m^{m \cdot n}$ scelte (il numero di funzioni si ottiene dalla cardinalità del codominio elevato il dominio)
- q_1: gli stati iniziali possibili sono m
- F: per 2^m scelte (i possibili stati terminali non è nient'altro che il numero di sottoinsiemi definibili in Q, ovvero $|\mathcal{P}(Q)|$)

E quindi abbiamo $m^{mn} \cdot m \cdot 2^m = m^{mn+1} \cdot 2^m$ possibili automi.

4) Mostrare che esiste un linguaggio regolare non accettato da qualsiasi automa a stati finiti con un solo stato di accettazione. [Corretto dal prof]



Sia $L = \{a^n | n \geq 0\} \cup \{b\}$, in questo caso è evidente che essendo lo stato iniziale di accettazione e che sia a che b devono essere accettati, necessariamente non può esistere un DFA con un solo stato di accettazione che accetti il linguaggio così descritto.

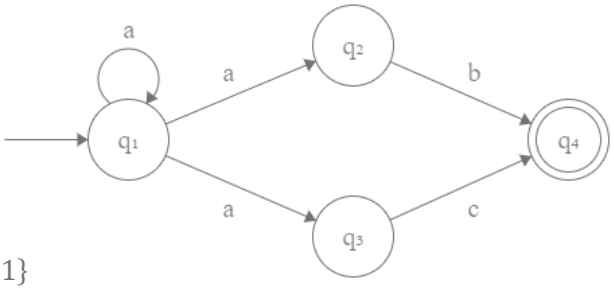
5) Sia $\mathcal{M}$ un automa nell'alfabeto $A = \{s_1, \dots, s_n\}$ con stati $Q = \{q_1, \dots, q_m\}$, funzione di transizione δ, stato iniziale q_1 e stati terminali F. Fornire una macchina di Turing $\mathcal{M}'$ cha accetta $L(\mathcal{M})$. [Corretto dal Prof]

Sia $L = L(\mathcal{M})$, con $\mathcal{M} = (Q, A, \delta, q_1, F)$. Allora $\mathcal{M}' = (Q \cup \{q_0\}, q_0, A, I)$ (l'aggiunta del nuovo stato iniziale è necessaria per descrivere quando la macchina di Turing è attualmente vuota).
Con $I = \{(q_0, \sqcup, R, q_1)\} \cup \{(q, a, R, \delta(q, a)) | q \in Q \wedge a \in A\} \cup \{(q, \sqcup, R, q) | q \in Q \setminus F\}$ (l'ultima istruzione è per descrivere gli stati non terminali, ovvero i loop nella macchina di Turing). La macchina di Turing così descritta accetta lo stesso linguaggio dell'automa: $L(\mathcal{M}') = L$

6) Descrivere il linguaggio accettato dal seguente NDFA con stato iniziale q_1

δ_1	a	b	c
q_1	$\{q_1, q_2, q_3\}$	$\varnothing$	$\varnothing$
q_2	$\varnothing$	$\{q_4\}$	$\varnothing$
q_3	$\varnothing$	$\varnothing$	$\{q_4\}$
q_4	$\varnothing$	$\varnothing$	$\varnothing$

$F_1 = \{q_4\}$.



$$L = \{a^n ab | n \geq 1\} \cup \{a^n ac | n \geq 1\}$$

15. Proprietà di chiusura e automi nonrestarting

Complemento di linguaggi regolari e altre proprietà

Teorema 1: Ogni linguaggio regolare è ricorsivamente enumerabile

Teorema 2 (chiusura rispetto il complemento): Se $L \subseteq A^*$ è regolare, anche $\overline{L} = A^* \setminus L$ è regolare.
Dimostrazione: Sia $\mathcal{M} = (Q, A, \delta, q_1, F)$ un DFA che accetta L, allora è evidente che scambiare gli stati di accettazione con quelli di non accettazione $(Q \setminus F)$ significa che una parola sarà accettata dal nuovo automa solo se non veniva accettata dall'automa precedente, ovvero: $\overline{\mathcal{M}} = (Q, A, \delta, q_1, Q \setminus F)$ accetta $\overline{L}$.

Corollario 3: L regolare ⇒ L ricorsivo.
Dimostrazione: Poiché L è regolare e lo è anche il suo complemento (per il teorema 2) sono entrambi ricorsivamente enumerabili (per il teorema 1), allora L è ricorsivo.

Teorema 4: L regolare $\Leftrightarrow L = L(\mathcal{M})$ per qualche NDFA $\mathcal{M}$.
Dimostrazione: Implicazione $\Rightarrow$: Sia $\mathcal{M}' = (Q, A, \delta', q_1, F)$ tale che $L = L(\mathcal{M}')$. Allora $\mathcal{M} = (Q, A, \delta, q_1, F)$ è un NDFA che descrive il precedente DFA con $\delta(q, a) = \{\delta'(q, a)\}$ per $q \in Q$ e $a \in A$, ed è un NDFA che accetta lo stesso linguaggio del DFA, quindi $L = L(\mathcal{M})$. Implicazione $\Leftarrow$: Sia $\mathcal{M} = (Q, A, \delta, q_1, F)$ un NDFA che accetta L. Definiamo un DFA $\mathcal{M}' = (\mathcal{P}(Q), A, \delta', \{q_1\}, F')$ con $F' = \{Q' \subseteq Q | Q' \cap F \neq \emptyset\}$ (l'insieme di stati che hanno intersezione non vuota con gli stati terminali dell'NDFA) e $\delta'(Q', a) = \bigcup_{q \in Q'} \delta(q, a)$ per $Q' \subseteq Q$ e $a \in A$ (l'unione al variare da $q \in Q'$ degli insiemi raggiunti a partire da q leggendo a). Da questa definizione si può dimostrare (noi non lo faremo) che $\forall w \in A^*, \forall Q' \subseteq Q, \delta'^*(Q', a) = \bigcup_{q \in Q'} \delta^*(q, w)$; allora $L(\mathcal{M}') = \{w \in A^* | \delta'^*(\{q_1\}, w) \in F'\} = \left\{w \in A^* \middle| \bigcup_{q \in Q'} \delta^*(q_1, w) \in F'\right\} = \{w \in A^* | \delta^*(q_1, w) \in F'\}$ $= \{w \in A^* | \delta^*(q_1, w) \cap F \neq \emptyset\} = L(\mathcal{M})$ per definizione. Come volevasi dimostrare.

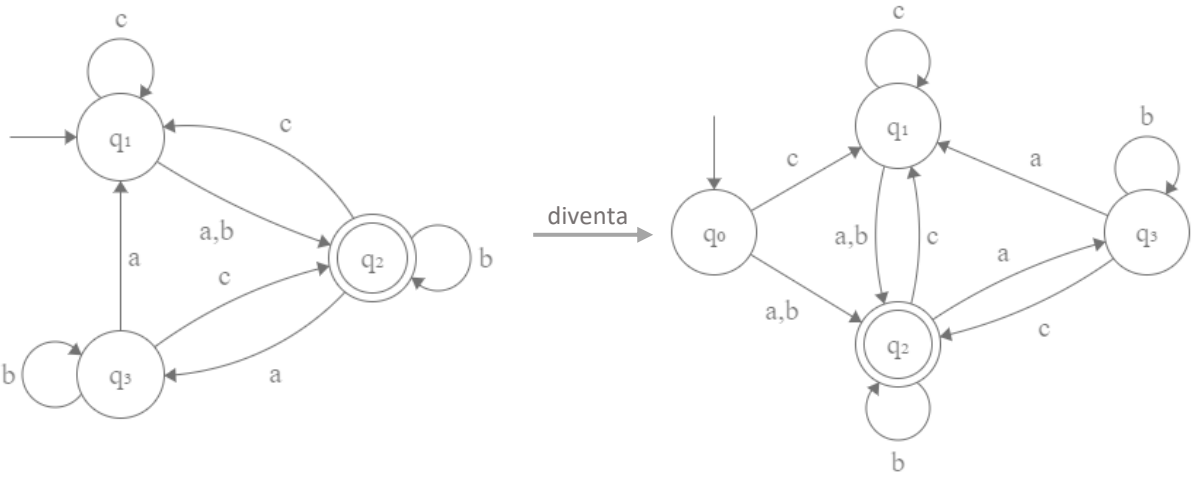
Automi nonrestarting

Un DFA $\mathcal{M}$(questa definizione si può generalizzare anche per un NDFA) è nonrestarting (non ricomincia) se non ci sono transizioni entranti nello stato iniziale (quindi quando lasciato non posso più tornare nello stato iniziale). Formalmente, $\mathcal{M}$ è **nonrestarting** se $\mathcal{M} = (Q, A, \delta, q_1, F)$ e $q_1 \notin \delta(Q \times A)$.

Dalla seguente proposizione possiamo osservare che la proprietà di nonrestarting non è un vincolo in più da richiedere, rispetto ad un normale DFA (o NDFA):

Proposizione: Se $L \subseteq A^*$ è regolare, esiste un DFA nonrestarting che lo accetta.
Dimostrazione: Sia $\mathcal{M} = (Q, A, \delta, q_1, F)$ un DFA tale che $L = L(\mathcal{M})$. Basta aggiungere un nuovo stato iniziale (ovviamente q_1 non lo sarà più), infatti, sia $q_0 \notin Q$ e definiamo $\mathcal{M}' = (Q \cup \{q_0\}, A, \delta', q_0, F')$, dove $\delta'(q, a) = \delta(q, a)$ se $q \in Q$ e $a \in A$; mentre, $\delta'(q_0, a) = \delta(q_1, a)$ $\forall a \in A$. Infine, gli stati di accettazione sarà l'insieme $F' = \begin{cases} F & se\ q_1 \notin F \quad (\text{se lo stato iniziale non era terminale}) \\ F \cup \{q_0\}\ altrimenti & (\text{bisogna accettare anche la stringa vuota}) \end{cases}$ ed è evidente che l'automa è nonrestarting e che accetta lo stesso linguaggio: $L(\mathcal{M}') = L$. Come volevasi dimostrare.

Esempio trasformazione di DFA in DFA nonrestarting (useremo l'automa dell'esercizio 2b):



Unione e intersezione di linguaggi regolari

Teorema (chiusura rispetto l'unione): Se $L, L' \subseteq A^*$ sono regolari, allora $L \cup L'$ è regolare.
Dimostrazione: Siano $\mathcal{M} = (Q, A, \delta, q_1, F)$ e $\mathcal{M}' = (Q', A, \delta', q_1', F')$ due DFA nonrestarting tali che $L = L(\mathcal{M})$ e $L' = L(\mathcal{M}')$ e $Q \cap Q' = \emptyset$ (necessario che siano disgiunti). Da questi due DFA definiamo un NDFA $\widehat{\mathcal{M}} = \left(Q \cup Q' \cup \{q_0\} \setminus \{q_1, q_1'\}, A, \widehat{\delta}, q_0, \widehat{F}\right)$ con $q_0 \notin Q \cup Q'$ e $\widehat{\delta}(q, a) = \begin{cases} \{\delta(q, a)\}\ se\ q \in Q \setminus \{q_1\} \\ \{\delta'(q, a)\}\ se\ q \in Q' \setminus \{q_1'\} \\ \{\delta(q_1, a), \delta'(q_1', a)\}\ se\ q = q_0 \end{cases}$
(nota che abbiamo tolto i vecchi stati iniziale poiché inutili, essendo gli automi precedenti nonrestarting), e

$\hat{F} = \begin{cases} F \cup F' & se\ q_1 \notin F\ e\ q_1' \notin F' \\ F \cup F' \cup \{q_0\} & altrimenti \end{cases}$ (quindi anche se solo uno dei due era terminale). Ora è evidente che $\widehat{\mathcal{M}}$ accetta $L \cup L'$ come volevasi dimostrare.

Corollario (chiusura rispetto l'intersezione): L, L' regolari ⇒ $L \cap L'$ regolare.
Dimostrazione: Basta osservare (per le leggi di De Morgan) che $L \cap L' = A^* \setminus \big((A^* \setminus L) \cup (A^* \setminus L')\big)$ e ricordare che la famiglia dei linguaggi regolari su A è chiusa rispetto a unione e complemento.

Linguaggi regolari: Proposizioni

Proposizione 1: il linguaggio vuoto ∅ è regolare.
Dimostrazione: Basta considerare qualunque DFA con $F = \emptyset$ che ovviamente accetterà il linguaggio vuoto.

Proposizione 2: $\forall w \in A^*$, il linguaggio (contenente una sola parola) $\{w\}$ è regolare.
Dimostrazione: Sia $w = a_1 a_2 \dots a_n$, con $a_i \in A$ per $1 \leq i \leq n$ e consideriamo un NDFA che accetta come parola solo w (unica parola accettata dal linguaggio), il cui diagramma è:

Formalmente $\mathcal{M} = (Q, A, \delta, q_0, F)$ con $Q = \{q_0, \dots, q_n\}$, $F = \{q_n\}$ e $\delta(q_i, a) = f(x) = \begin{cases} \{q_{i+1}\}\ se\ a = q_{i+1} \\ \emptyset \quad altrimenti \end{cases}$.

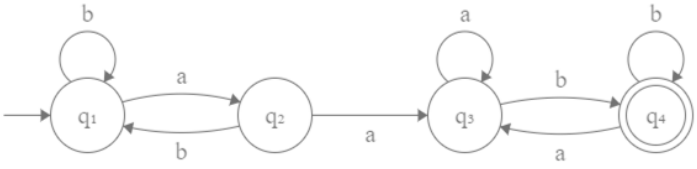
Corollario 3: Tutti i linguaggi finiti sono regolari.

Operatore Prodotto (concatenazione)

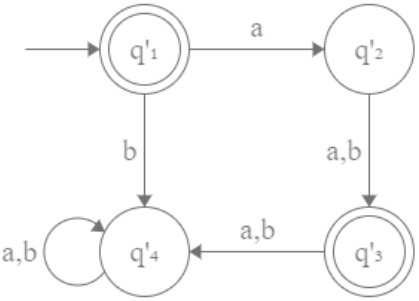
Dati $L, L' \subseteq A^*$, definiamo il prodotto di due linguaggi regolari l'insieme $LL' = \{uv \in A^* | u \in L \wedge v \in L'\}$.

Teorema (chiusura rispetto il prodotto): Se $L, L' \subseteq A^*$ sono regolari, lo è anche LL'.
Dimostrazione: Siano $\mathcal{M} = (Q, A, \delta, q_1, F)$ e $\mathcal{M}' = (Q', A, \delta', q_1', F')$ due DFA tale che $Q \cap Q' = \emptyset$ e siano $L(\mathcal{M}) = L$ e $L(\mathcal{M}') = L'$. Prendiamo come insiemi di stati $L \cup L'$ a cui aggiungiamo delle transizioni ma mantenendo invariato l'insieme degli stati (quindi non sarà più deterministico); più precisamente l'NDFA $\widehat{\mathcal{M}} = \big(Q \cup Q', A, \hat{\delta},\ q_1, \hat{F}\big)$, con $\hat{F} = \begin{cases} F'\ se\ q_1' \notin F' \\ F \cup F'\ altrimenti \end{cases}$ e $\hat{\delta}\,(q, a) = \begin{cases} \{\delta(q,a)\} & se\ q \in Q \setminus F \\ \{\delta'(q,a)\} & se\ q \in Q' \\ \{\delta(q,a), \delta'(q_1', a)\}\ se\ q \in F \end{cases}$.
Le parole accettate da questo automa è chiaro siano quelle costituite da una parola accettata dal primo seguita da una parola accettata del secondo, ovvero accetta LL', come volevasi dimostrare.

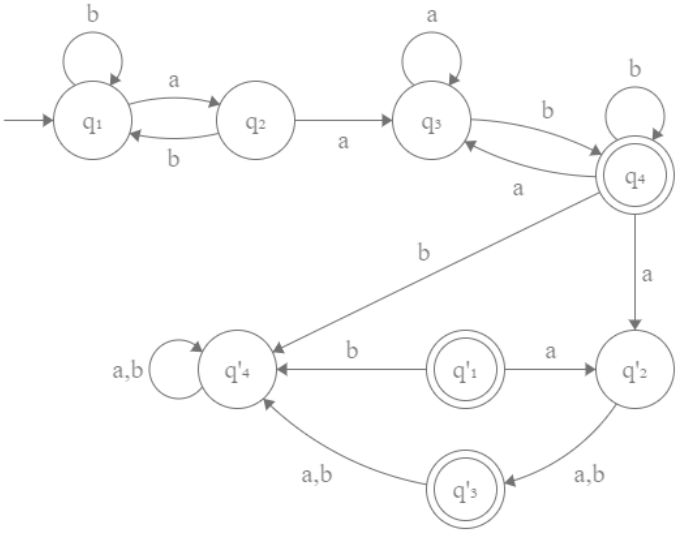
Esempio: Sia $A = \{a, b\}$ e $L = \{w \in A^* | w \text{ contiene } aa \text{ e finisce per } b\}$, $L' = \{\varepsilon, aa, ab\}$.
L'automa che accetta L sarà:



Mentre, l'automa che accetta L' è:



E dunque, il prodotto LL' è accettato dal seguente NDFA:

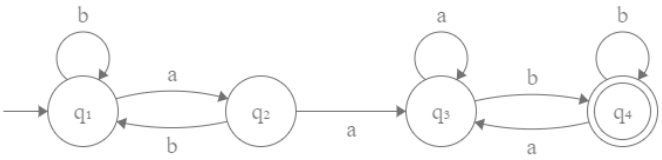


Operatore Stella

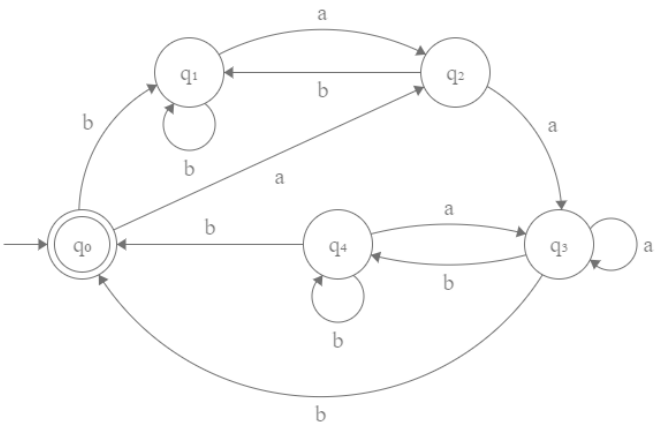
Dato $L \subseteq A^*$, definiamo $L^* = \{u_1 u_2 \dots u_n \in A^* | n \geq 0 \wedge u_i \in L\}$. Nota: possiamo definire le potenze L^n nel seguente modo: $L^0 = \{\varepsilon\}$ e, per $n > 0, L^{n+1} = L \cdot L^n$. Allora si ha $L^* = \bigcup_{n=0}^{\infty} L^n$.

Teorema (chiusura rispetto l'operazione stella): $L \subseteq A^*$ regolare ⇒ L^* regolare.
Dimostrazione: Partiamo da un DFA e costruiamo un NDFA che accetti L^* come fatto con il prodotto. Sia $\mathcal{M} = (Q, A, \delta, q_1, F)$ un DFA nonrestarting che accetta L, e consideriamo l'NDFA $\widehat{\mathcal{M}} = \left(Q, A, \widehat{\delta}, q_1, \{q_1\}\right)$, con $\widehat{\delta}(q,a) = \begin{cases} \{\delta(q,a)\} \; se \; \delta(q,a) \notin F \\ \{\delta(q,a), q_1\} \; altrimenti \end{cases}$ (ricopiamo nello stato iniziale tutte le transizioni che andavano in stati di accettazione). Mostriamo che $\forall n$, $\widehat{\mathcal{M}}$ accetta concatenazioni di n parole di L, per induzione su n. Caso base: $n = 0$ è ovvio poiché la parola vuota è sempre accettata. Caso Ricorsivo: supponiamo vero che $\widehat{\mathcal{M}}$ accetta concatenazioni di $n-1$ parole di L e mostriamo che accetta anche quelle di n parole di L; infatti, dopo aver letto le prime $n-1$ parole, l'insieme degli stati attuali contiene q_1, e dunque, per come abbiamo definito $\widehat{\mathcal{M}}$, lo stesso avviene dopo averne letta un'altra (l'n-esima). Questo mostra che sicuramente il linguaggio accettato da questo automa contiene tutte le parole L^n e quindi $L^* \subseteq L\left(\widehat{\mathcal{M}}\right)$. D'altra parte, poiché $\mathcal{M}$ era nonrestarting , l'unico modo di tornare a q_1 in $\widehat{\mathcal{M}}$ è leggendo parole di L, quindi $L(\mathcal{M}) \subseteq L^*$ e dunque $L\left(\widehat{\mathcal{M}}\right) = L^*$, come volevasi dimostrare.

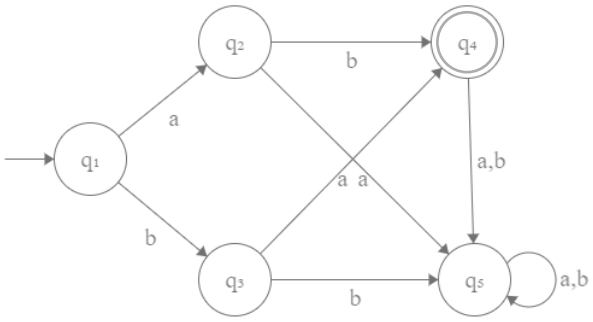
Esempio 1: Sia $A = \{a, b\}$ e $L = \{w \in A^* | w \text{ contiene } aa \text{ e finisce per } b\}$, accettato dal seguente automa:

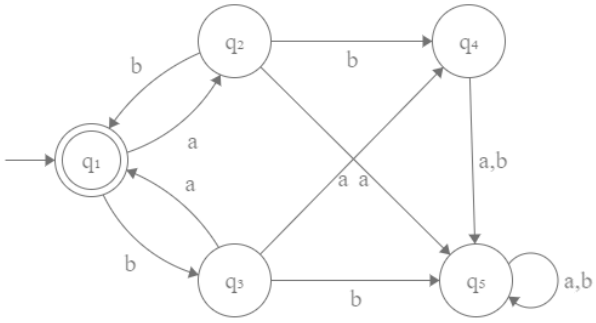


si ha allora, ad esempio, che $baababaabb \in L^*$.
In realtà $LL \subseteq L$, per cui $L^* = L \cup \{\varepsilon\}$ **in questo caso particolare**, ma applichiamo ugualmente la dimostrazione del teorema, rendendo prima l'automa nonrestarting e costruendo in seguito l'NDFA:



Esempio 2: Sia $L = \{ab, ba\}$ e costruiamo un NDFA che accetti L^*. Partiamo da un DFA che accetta L che è già nonrestarting, dunque risulterà: Che diventa:





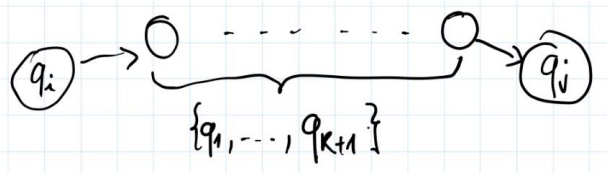
16. Espressioni regolari e Teorema di Kleene

Teorema di Kleene

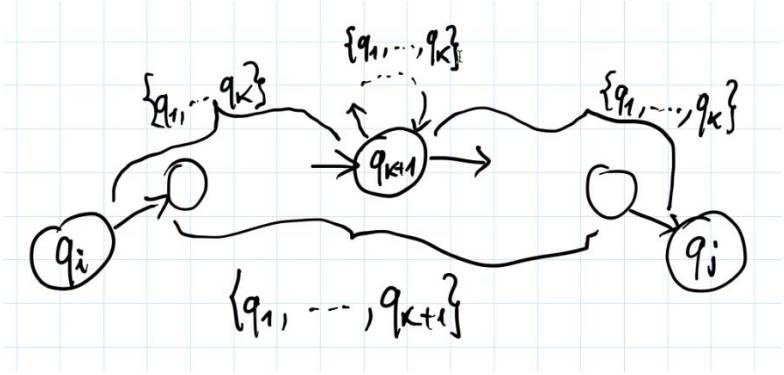
Teorema: Un linguaggio $L \subseteq A^*$ è regolare se e solo se è finito o si ottiene da linguaggi finiti attraverso un numero finito di applicazioni delle operazioni regolari $\cup$, $\cdot$, $*$ (unione, concatenazione e stella).

Dimostrazione: Implicazione $\Leftarrow$: immediata dalle proprietà di chiusura viste finora e poiché i linguaggi finiti sono regolari. Implicazione $\Rightarrow$: Sia $\mathcal{M} = (Q, A, \delta, q_1, F)$ un DFA che accetta L, con $Q = \{q_1, \dots, q_n\}$. Definiamo, per $1 \leq i, j \leq n$ e $k \leq n$, un insieme di parole da $R_{i,j}^{(k)} = \{w \in A^* | \delta^*(q_i, w) = q_j \text{ e } \delta^*(q_i, w') \in \{q_1, \dots, q_k\} \text{ se } w' \text{ è un prefisso non vuoto e proprio}(\neq w) \text{ di } w\}$ (partendo da q_i arrivo a q_j e gli stati intermedi sono compresi nell'insieme q_1 fino a q_k, quindi da q_i a q_j non attraverso gli stati da q_{k+1} a q_n). In particolare, per $k = 0$ otteniamo l'insieme $R_{i,i}^{(0)} = \{\varepsilon\} \cup \{a \in A | \delta(q_i, a) = q_i\}$ (finito), e per $j \neq i$ si ha $R_{i,j}^{(0)} = \{a \in A | \delta(q_i, a) = q_j\}$ (finito). Inoltre, per $k \geq 0$ si ha: $R_{i,j}^{(k+1)} = R_{i,j}^{(k)} \cup R_{i,k+1}^{(k)} \cdot \left(R_{k+1,k+1}^{(k)}\right)^* \cdot R_{k+1,j}^{(k)}$. Dunque, tutti gli insiemi $R_{i,j}^{(k)}$ si ottengono da linguaggi finiti usando un numero finito di volte le operazioni $\cup$, $\cdot$ e $*$. Ma $L = L(\mathcal{M}) = \bigcup_{j \in F} R_{1,j}^{(n)}$, dunque anche L si ottiene allo stesso modo, come volevasi dimostrare.

Questo teorema da una caratterizzazione dei linguaggi regolari, cerchiamo di dare una migliore spiegazione della relazione $R_{i,j}^{(k+1)} = R_{i,j}^{(k)} \cup R_{i,k+1}^{(k)} \cdot \left(R_{k+1,k+1}^{(k)}\right)^* \cdot R_{k+1,j}^{(k)}$ visualizzandola sul diagramma dell'automa:



L'equazione dice semplicemente che un percorso del genere non passa mai per lo stato $k+1$ e quindi siamo in $R_{i,j}^{(k)}$ (primo caso), altrimenti si considera l'insieme $R_{i,k+1}^{(k)} \cdot \left(R_{k+1,k+1}^{(k)}\right)^* \cdot R_{k+1,j}^{(k)}$, ovvero:



Espressioni regolari

Supponiamo di avere un alfabeto finito $A = \{a_1, \dots, a_k\}$ e poniamo $\widehat{A} = \left\{\underline{a_1}, \dots, \underline{a_k}, \underline{\emptyset}, \underline{\varepsilon}, \underline{\cup}, \underline{\cdot}, \underline{*}, \underline{(}, \underline{)}\right\}$ (le sottolineature verranno dimenticate quanto prima). Allora un'espressione regolare su A è una particolare stringa sull'alfabeto esteso $\widehat{A}^*$ appartenente ad un particolare sottoinsieme definito dalle seguenti regole (**Sintassi**):

1) I simboli $\underline{a_1}, \dots, \underline{a_k}, \underline{\emptyset}$ e $\underline{\varepsilon}$ sono espressioni regolari
2) Se α, β sono espressioni regolari, lo è anche $(\alpha \,\underline{\cup}\, \beta)$;
3) Se α, β sono espressioni regolari, lo è anche $(\alpha \,\underline{\cdot}\, \beta)$
4) Se α è una espressione regolare, lo è anche $\alpha\underline{^*}$;

Ad ogni espressione regolare α su A corrisponde un linguaggio $\langle\alpha\rangle \subseteq A^*$, secondo le seguenti regole (**Semantica):**

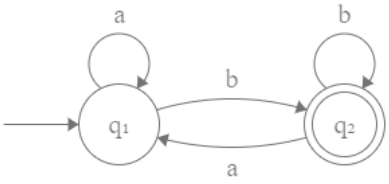
1) Per $i = 1, \dots, k$, $\langle\underline{a_i}\rangle = \{a_i\}$
2) $\langle\underline{\emptyset}\rangle = \emptyset$ e $\langle\underline{\varepsilon}\rangle = \varepsilon$
3) Se α e β sono espressioni regolari, $\langle(\alpha \,\underline{\cup}\, \beta)\rangle = \langle\alpha\rangle \cup \langle\beta\rangle$
4) Se α e β sono espressioni regolari, $\langle(\alpha \,\underline{\cdot}\, \beta)\rangle = \langle\alpha\rangle \cdot \langle\beta\rangle$
5) Se α è espressione regolare, $\langle\alpha\underline{^*}\rangle = \langle\alpha\rangle^*$

A questa definizione formale, nella pratica si usa semplificare la scrittura delle espressioni regolare dando delle convenzioni sulla precedenza delle operazioni, così da poter eliminare parentesi e " · ". Le regole di precedenza delle operazioni: $\underline{*} > \underline{\cdot} > \underline{\cup}$ (da questo momento in poi la sottolineatura verrà dimenticata).
Suggerimento: praticamente puoi considerare le regole di precedenza matematiche considerando la stella come una potenza, la concatenazione come una moltiplicazione e l'operatore unione come una somma.

Per il teorema di Kleene, un linguaggio $L \subseteq A^*$ è regolare se e solo se esiste un'espressione regolare α (ovviamente sullo stesso alfabeto) tale che $\langle\alpha\rangle = L$.

Esempio 1: Sia L il linguaggio formato dalle parole $\{a, b\}$ che contengono aa e terminano in b. Proviamo a scrivere un'espressione regolare sull'alfabeto $\{a, b\}$. Allora $L = \langle\left(\left(\left((a \cup b)^* \cdot (a \cdot a)\right) \cdot (a \cup b)^*\right) \cdot b\right)\rangle$, che può essere riscritto in versione semplificata come $L = \langle(a \cup b)^* aa (a \cup b)^* b\rangle$

Esempio 2 (Applicazione del teorema di Kleene per analizzare un DFA):

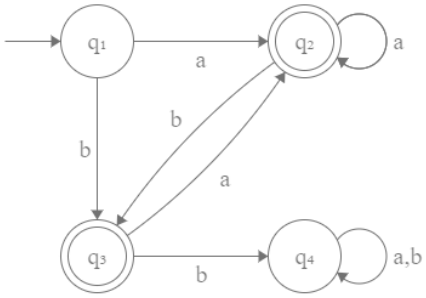


$L(\mathcal{M}) = R_{1,2}^{(2)} = R_{1,2}^{(1)} \cup R_{1,2}^{(1)} \cdot \left(R_{2,2}^{(1)}\right)^* \cdot R_{2,2}^{(1)}$ dove $R_{1,2}^{(1)} = \langle a^*b \rangle$ (tutto quello che leggo da q_1 per arrivare a q_2 senza ripassare per q_2), $R_{2,2}^{(1)} = \langle \varepsilon \cup b \cup aa^*b \rangle$ e quindi risulta $L(\mathcal{M}) = \langle \underbrace{a^*b}_{R_{1,2}^{(1)}} \cup \underbrace{a^*b}_{R_{1,2}^{(1)}} \underbrace{(\varepsilon \cup b \cup aa^*b)^*}_{\left(R_{2,2}^{(1)}\right)^*} \underbrace{(\varepsilon \cup b \cup aa^*b)}_{R_{2,2}^{(1)}} \rangle$.

Possiamo ulteriormente semplificare il linguaggio utilizzando la proprietà distributiva: $L_1L_2 \cup L_1L_3 = L_1(L_2 \cup L_3)$ e un'altra utile proprietà: $L_1^*L_1 \cup \{\varepsilon\} = L_1^*$ (queste semplificazioni risultano intuitivamente più semplici se si pensa alle operazioni come somma, prodotto e potenza e come elemento neutro $\varepsilon(= 1)$. Quindi praticamente sopra mettiamo a fattor comune a^*b, risulterà dunque $L(\mathcal{M}) = \langle a^*b(\varepsilon \cup (\varepsilon \cup b \cup aa^*b)^*(\varepsilon \cup b \cup aa^*b)) \rangle = \langle a^*b(\varepsilon \cup b \cup aa^*b)^* \rangle = \langle a^*b(b \cup aa^*b)^* \rangle$.

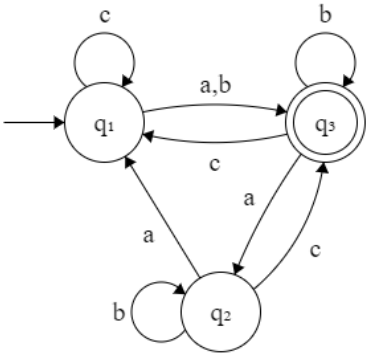
Esercizi Svolti

1) Sia $L = \{x \in \{a,b\}^* | x \notin \varepsilon \text{ e } bb \text{ non è sottostringa di } x\}$,
   a. Mostrare che L è regolare costruendo un DFA $\mathcal{M}$ tale che $L = L(\mathcal{M})$



   b. Trovare un'espressione regolare α tale che $L = \langle \alpha \rangle$
   $L = \langle b(a \cup ab)^* \cup (a \cup ab)(a \cup ab)^* \rangle = \langle (b \cup a \cup ab)(a \cup ab)^* \rangle$

2) Dato il seguente automa $\mathcal{M}$ usare il teorema di Kleene per definire il linguaggio accettato da $\mathcal{M}$:



$L(\mathcal{M}) = R_{1,3}^{(3)} = R_{1,3}^{(2)} \cup R_{1,3}^{(2)} \cdot \left(R_{3,3}^{(2)}\right)^* \cdot R_{3,3}^{(2)}$ dove $R_{1,3}^{(2)} = R_{1,3}^{(1)} = \langle c^*(a \cup b) \rangle$ e $R_{3,3}^{(2)} = R_{3,3}^{(1)} \cup R_{3,2}^{(1)} \cdot \left(R_{2,2}^{(1)}\right)^* \cdot R_{2,3}^{(1)} = \langle \varepsilon \cup b \cup cc^*(a \cup b) \cup a(\varepsilon \cup b)^*\big(c \cup ac^*(a \cup b)\big) \rangle = \langle \varepsilon \cup b \cup ab^*c \cup (c \cup ab^*a)c^*(a \cup b) \rangle = \langle \alpha \rangle$ dunque $L(\mathcal{M}) = \langle c^*(a \cup b)(\varepsilon \cup \alpha^*\alpha) \rangle = \langle c^*(a \cup b)\alpha^* \rangle$, e sostituendo α avremo il risultato:

$$L(\mathcal{M}) = \langle c^*(a \cup b)\big(b \cup ab^*c \cup (c \cup ab^*a)c^*(a \cup b)\big)^* \rangle$$

17. Pumping lemma per linguaggi regolari e conseguenze

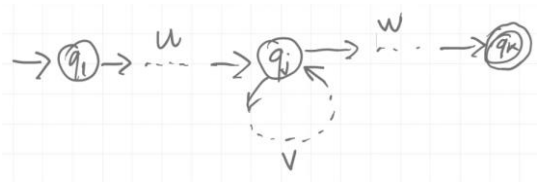
Pumping lemma (lemma di iterazione)

Lemma di iterazione (pumping lemma): Sia $\mathcal{M}$ un DFA su A con n stati, e sia $x \in L(\mathcal{M}) : |x| \geq n$. Allora $\exists u, v, w \in A^*$ tale che:

1) $x = uvw$
2) $v \neq \varepsilon$
3) $\forall i \geq 0, uv^iw \in L(\mathcal{M})$

Dimostrazione: Usiamo il "principio della piccionaia": se abbiamo un numero $\geq n + 1$ di oggetti da distribuire in n insiemi, allora almeno uno di tali insiemi contiene almeno 2 oggetti. Dato che $|x| \geq n$, il numero totale di passaggi per uno stato lungo il cammino etichettato con x in $\mathcal{M}$ (a partire dallo stato iniziale) è almeno $n + 1$; dunque esiste almeno uno stato per cui passiamo almeno 2 volte (essendo n il

numero totale degli stati). Se $Q = \{q_1, \dots, q_n\}$ è l'insieme degli stati di $\mathcal{M}$,abbiamo che $\delta^*(q_1, x) = q_k \in F$, e se $x = x_1 \dots x_m$ con $m \geq n$ allora $\exists h_1, h_2 : 1 \leq h_1 < h_2 \leq m$ e $\delta^*(q_1, x_1 \dots x_{h_1}) = q_j = \delta^*(q_1, x_1 \dots x_{h_2})$ (questa è praticamente la formalizzazione di "esiste uno stato da cui passiamo almeno due volte):



Siano allora $u = x_1 \dots x_{h_1}, v = x_{h_1+1} \dots x_{h_2}, w = x_{h_2+1} \dots x_m$, è evidente che $x = uvw$ ed inoltre che $v \neq \varepsilon$ dato che $h_1 < h_2$ (ed anche perché stiamo passando due volte per q_j). Rimane quindi da dimostrare solo che $\forall i \geq 0, uv^i w \in L(\mathcal{M})$, ma ciò è evidente poiché $\forall i \geq 0, \delta^*(q_i, uv^i w) = q_k$, per cui $uv^i w \in L(\mathcal{M})$.
Osservazione: questo lemma esprime una condizione **necessaria** affinché un dato linguaggio sia regolare.
N.B.: la terza condizione del lemma si può scrivere anche come $\langle uv^*w \rangle = \{u\}\{v\}^*\{w\} \subseteq L(\mathcal{M})$.

Conseguenze del pumping lemma

Corollario: Sia $\mathcal{M}$ un DFA su A con n stati. Se $L(\mathcal{M}) \neq \emptyset$, allora $L(\mathcal{M})$ contiene parole di lunghezza $< n$.
Dimostrazione: Se per assurdo $\mathcal{M}$ accettasse solo parole di lunghezza $\geq n$, prendendo una parola di lunghezza minima $x \in L(\mathcal{M})$, dal pumping lemma avremmo $x = uvw$ con $v \neq \varepsilon$ e $uw \in L(\mathcal{M})$ ma ciò è assurdo poiché x è di lunghezza minima e quindi $|uw| < |x|$, come volevasi dimostrare.

Questo corollario ci consente di definire un algoritmo che in tempo finito ci dice se un automa accetta parole oppure no (con gli automi visti finora il problema non si poneva, essendo infatti di pochi stati bastava guardarlo). Non ci serve controllare tutti i possibili percorsi di un automa ma ci basta controllare che di tutte le parole di lunghezza inferiore al numero degli stati ce ne sia almeno una accettata.

Possiamo usare questo risultato anche per verificare, dati due DFA $\mathcal{M}_1, \mathcal{M}_2$, se $L(\mathcal{M}_1) \subseteq L(\mathcal{M}_2)$: basta costruire un DFA che accetti $L(\mathcal{M}_1) \setminus L(\mathcal{M}_2) = L(\mathcal{M}_1) \cap \overline{L(\mathcal{M}_2)}$; se esso non accetta parole (per verificarlo basta verificare che non accetti parole di lunghezza inferiore al numero dei suoi stati) allora significa che tale differenza è vuota e in altre parole, che il primo linguaggio è incluso nel secondo.

Proposizione: Sia $\mathcal{M}$ un DFA con n stati.
Allora $L(\mathcal{M})$ è infinito $\Leftrightarrow$ $L(\mathcal{M})$ contiene parole x di lunghezza $n \leq |x| < 2n$.
Dimostrazione: Implicazione $\Leftarrow$: Se $x \in L(\mathcal{M})$ e $n \leq |x| < 2n$, per il pumping lemma $x = uvw$ con $v \neq \varepsilon$ e $uv^i w \in L(\mathcal{M})$ per $i \geq 0$, quindi $L(\mathcal{M})$ è infinito. Implicazione $\Rightarrow$: Sia $x \in L(\mathcal{M})$ la più corta parola con lunghezza di almeno $2n$ (esisterà poiché il linguaggio accettato è infinito), dunque $|x| \geq 2n$ ma più corta possibile. Decomponiamo x in due parti $x = yz$ con $|y| = n$ e, dunque, $|z| \geq n$. Applicando il principio della piccionaia come nel pumping lemma (non possiamo applicare direttamente il lemma poiché y potrebbe non essere accettata come parola), otteniamo che $y = uvw$ con $v \neq \varepsilon$ e $\forall i \geq 0 \quad uv^i wz \in L(\mathcal{M})$ e in particolare, per $i = 0$ si ha che $uwz \in L(\mathcal{M})$. Dunque, $|uwz| \geq |z| \geq n$ e, inoltre, $|uwz| \leq 2n$ poiché $|uwz| < |uvwz| = |x|$ e ciò va contro alla nostra definizione di x (la parola più corta di lunghezza $\geq 2n$).

Esempi di come usare il pumping lemma

1. Il linguaggio $L = \{a^n b^n | n \geq 0\}$ non è regolare. Intuitivamente, perché un eventuale DFA che lo accetti dovrebbe avere infiniti stati, una chiara contraddizione. Per una dimostrazione formale possiamo usare il pumping lemma. Se per assurdo L fosse regolare, sarebbe accettato da un DFA con $p \geq 1$ stati. La parola $x = a^p b^p$ dovrebbe verificare la condizione $x = uvw$ con $v \neq \varepsilon$ e $uv^i w \in L$ per $i \geq 0$. Ma i casi per la decomposizione $x = uvw$ con $v \neq \varepsilon$ sono:
    1) $v = a^j$ per $j > 0$, ed in questo caso $uv^0 w = a^{p-j} b^p \notin L(\mathcal{M})$
    2) $v = b^j$ per $j > 0$, ed in questo caso $uv^0 w = a^p b^{p-j} \notin L(\mathcal{M})$
    3) $v = a^j b^k$ per $j, k > 0$ ed in questo caso $uv^2 w = a^{p-j}\left(a^j b^k\right)^2 b^{p-k} = a^p b^k a^j b^p \notin L(\mathcal{M})$

2. Vediamo ora un esempio un po' meno banale, consideriamo il linguaggio $L = \{a^n b^m | n \geq m \geq 0\}$, anche questo linguaggio non è regolare. Sia per assurdo $\mathcal{M}$ un DFA con p stati tale che $L = L(\mathcal{M})$. Scegliamo ancora la parola $x = a^p b^p$; valgono gli stessi casi di prima per la decomposizione $x = uvw$. Nel primo caso $v = a^j$, per $j > 0$, consideriamo $uv^0w = uw = a^{p-j}b^p \notin L$. Gli altri due casi possono essere trattati come prima (si noti come sia stato essenziale scegliere il caso giusto per il primo caso).

3. Questo esempio ci mostra come a volte il pumping lemma non basti per dimostrare la regolarità di un linguaggio. Sia $L = \{x \in \{a, b\}^* | x = \tilde{x}\}$ (l'insieme di tutti i palindromi, ovvero quelli che si possono leggere in entrambi i sensi, $\tilde{x}$ rappresenta la parola letta da destra a sinistra). È intuitivo che questo linguaggio non è regolare per un problema di memoria, poiché non c'è modo di memorizzare le lettere lette per un automa a stati finiti. Non è possibile mostrare che questo linguaggio sia regolare usando il pumping lemma poiché qualsiasi palindromo non vuoto x, si può decomporre come $x = uvw$ dove $|v| \leq 2$ e $w = \tilde{u}$, e allora $\forall i \geq 0, uv^i w \in L$ (abbiamo una parola con una parte centrale che anche se la cancelliamo la nuova parola è comunque palindroma).
Ripensando alla dimostrazione del pumping lemma: Sia $\mathcal{M}$ un DFA con n stati e $x \in L(\mathcal{M})$ con $|x| \geq n$ con $x = x_1 x_2 \dots x_m$ e $m \geq n$. Poiché $\{\delta^*(q_1, x_1 \dots x_k) | 0 \leq k \leq m\}$ contiene al massimo n stati distinti, esistono $j_1 < j_2$ tali che $\delta^*(q_1, x_1 \dots x_{j_1}) = \delta^*(q_1, x_1 \dots x_{j_2})$ (esistono due prefissi della parola x che corrispondono allo stesso stato). Scegliamoli minimali, cioè: gli stati $\{\delta^*(q_1, x_1 \dots x_k) | 0 \leq k \leq j_2\}$ sono tutti diversi, per cui $j_2 \leq n$ e quindi, se poniamo $u = x_1 \dots x_{j_1}$, $v = x_{j_1+1} \dots x_{j_2}$ e $w = x_{j_2} \dots x_m$, allora:
1) $v \neq \varepsilon$
2) $\forall i \geq 0, uv^i w \in L(\mathcal{M})$
3) $|uv| \leq n$ (la parola centrale deve quindi stare all'interno delle prime n lettere)

Con queste nuove supposizioni non può più succedere il caso sopracitato in cui v era al centro del palindromo e quindi possiamo dimostrare che L non è regolare. Sia per assurdo $\mathcal{M}$ un DFA con p stati tale che $L = L(\mathcal{M})$ e consideriamo il palindromo $(ab)^p(ba)^p \in L$ e questa parola ha $|x| = 4p \geq p$. Consideriamo le decomposizioni $x = uvw$ con $v \neq \varepsilon$ e $|uv| \leq p$ (quindi non può stare al centro), $\{\delta^*(q_1, x_1 \dots x_k) | 0 \leq k \leq m\}\{\delta^*(q_1, x_1 \dots x_k) | 0 \leq k \leq m\}$schematizzando la parola:
$ab \underbrace{ab \dots}_{v} a\boldsymbol{bb}a \dots baba$. Quindi è evidente che la parola $uw \notin L$ (non è un palindromo) qualunque sia la scelta di v, poiché contiene bb nelle sue prime $2p$ lettere ma non nelle ultime $2p$ lettere. Dunque, il pumping lemma "esteso" non è verificato e quindi il DFA in questione non può esistere cioè il linguaggio non è regolare.

18. Grammatiche e linguaggi context-free

Grammatiche context-free

Nasce dallo studio di linguaggi naturali, più precisamente per analizzare come deve essere strutturata una frase in una certa lingua. Le grammatiche context-free (libere da contesto) sono note per l'analisi di linguaggi naturali (Chomsky e co.) adattate poi ai linguaggi formali (sottoinsiemi di A^* qualsiasi) e trovano tutt'oggi applicazione per descrivere la sintassi formale di un linguaggio di programmazione.

Formalmente, una grammatica context-free (**CFG**) è una quadrupla $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{\Sigma}, \boldsymbol{R}, \boldsymbol{S})$, dove
1) $\boldsymbol{V}$ è un **alfabeto** finito (di cui elementi vengono detti **variabili**)
2) $\boldsymbol{\Sigma}$ è un **alfabeto** finito (di cui elementi vengono detti **terminali**)
3) $\boldsymbol{R} \subseteq V \times (V \cup \Sigma)^*$ è un **insieme** di coppie i cui elementi sono detti **regole** o **produzioni**
4) $\boldsymbol{S} \in V$ è detta variabile iniziale o **assioma**

Una regola (X, u) si indica normalmente con $X \to u$ (X produce u). Date $u, w \in (V \cup \Sigma)^*$, diciamo che u **produce** w (per **derivazione elementare** secondo G) se R contiene una regola $X \to v$ tale che $u = u_1 X u_2$ e $w = u_1 v u_2$ per qualche $u_1, u_2 \in (v \cup \Sigma)^*$. Scriviamo $u \Rightarrow w$ (u produce w, se non è chiaro il contesto si usa

il simbolo $\underset{G}{\Rightarrow}$) per le derivazioni elementari (singole regole). Invece scriviamo $u \overset{*}{\Rightarrow} w'$ ($\overset{*}{\underset{G}{\Rightarrow}}$) se $\forall n \geq 0, w_1, w_2, \dots w_n$ tali che $u \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n = w'$. Diciamo allora che w' si ottiene da u per **derivazione** (secondo G). Allora il **linguaggio generato** sarà $L(G) = \left\{x \in \Sigma^* \middle| S \overset{*}{\Rightarrow} x\right\}$.

Esempio 1: Sia $G = (V, \Sigma, R, S)$ con $V = \{S\},\ \Sigma = \{a, b\},\ R = \{S \to aSb, S \to \varepsilon\}$ (di norma le regole di una grammatica si scrivono $S \to aSb|\varepsilon$ e si legge S produce aSb oppure ε). Allora $L(G) = \{a^n b^n | n \geq 0\}$. Infatti, una derivazione per $a^n b^n$ è: $S \Rightarrow aSb \Rightarrow aaSbb = a^2 Sb^2 \Rightarrow \cdots \Rightarrow a^n Sb^n \Rightarrow a^n b^n$ (utilizzando la seconda regola che sostituisce S con la parola vuota). Questo esempio ci dimostra anche che le grammatiche context-free sono in grado di generare linguaggi non regolari (vedi esempio 1 del capitolo precedente).

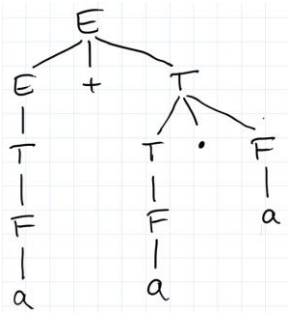
Esempio 2: Sia $G = (V, \Sigma, R, S)$ con $S \to aSb|\varepsilon$. Derivazione elementare: $u \Rightarrow v$ se $\exists X \to w \in R, \exists u_1, u_2 \in (V \cup \Sigma)^*: U = u_1 X u_2, v = u_1 w u_2$. Derivazione: sequenze di zero o più derivazioni elementari $\left(x \overset{*}{\Rightarrow} y\right)$ con $L(G) = \left\{x \in \Sigma^* \middle| S \overset{*}{\Rightarrow} x\right\}$. Sia $L = \{a^n b^m | n \geq m > 0\}$, possiamo scrivere $a^n b^m = a^{n-m} a^m b^m$. Siano le seguenti regole $S \to AX$, $A \to aA|\varepsilon$, $X \to aXb|ab$, è evidente che $a^4 b^3 \in L$, essendo $S \Rightarrow AX \Rightarrow aAX \Rightarrow aX \Rightarrow aaXb \Rightarrow aaaXbb \Rightarrow aaaabbb$. Nota: una regola come $S \to AX$ può essere interpretata come equazione su linguaggi; cioè $L = L_A L_X$, con $L_A = \{a^k | k \geq 0\}$ e $L_X = \{a^n b^m | m > 0\}$.

Alberi di derivazione, ambiguità

Consideriamo la grammatica G_1 con le seguenti regole: $E \to E + T \mid T,\ T \to T \cdot F \mid F,\ F \to (E) \mid a$ (+ e · sono dei terminali); dove $G_1 = (V, \Sigma, R, E)$ con $V = \{E, T, F\},\ \Sigma = \{a, +, \cdot, (,)\}$. Il linguaggio generato da questa grammatica è costituito da espressioni, cioè una somma di termini dove ogni termine è un prodotto di fattori ed un fattore è o semplicemente la variabile a oppure una qualsiasi espressione tra parentesi. Ad esempio, $a + a \cdot a \in L(G_1)$: $E \Rightarrow E + T \Rightarrow T + T \Rightarrow T + T \cdot F \Rightarrow T + F \cdot F \Rightarrow F + F \cdot F \Rightarrow a + F \cdot F \Rightarrow a + a \cdot F \Rightarrow a + a \cdot a$. Si noti che questa non è l'unica derivazione per la stringa $a + a \cdot a$, ma può cambiare solo l'ordine in cui applico le regole e non le regole stesse (esempio: $T + T \Rightarrow F + T \Rightarrow a + T \Rightarrow a + T \cdot F \Rightarrow a + F \cdot F \Rightarrow \cdots$) e questo rende le derivazioni equivalenti, per introdurre il concetto di ambiguità bisogna prima descrivere l'albero di derivazione.

Albero di derivazione: Data una derivazione in G, costruiamo un albero che ha la variabile iniziale come radice, e tale che a ogni passo della derivazione corrispondono tanti figli della variabile da sostituire quante sono le lettere dalla stringa che la sostituisce (i figli saranno nominati da sinistra a destra in base alla variabile da sostituire).

Per l'esempio descritto in precedenza avremo il seguente albero di derivazione per $a + a \cdot a \in L(G_1)$:

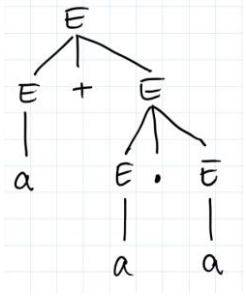


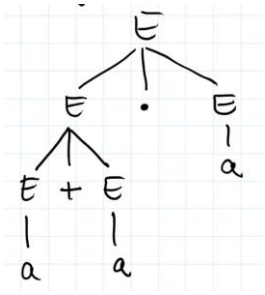
E questo sarà lo stesso albero anche per … $\Rightarrow T + T \Rightarrow F + T \Rightarrow a + T \Rightarrow a + T \cdot F \Rightarrow a + F \cdot F \Rightarrow \cdots$; di conseguenza questo è un buon metodo per non confondere derivazioni equivalenti con ambiguità.

Due derivazioni sono **equivalenti** se corrispondono allo stesso albero, o equivalentemente alla stessa derivazione **estrema a sinistra**, in cui a ogni passo la variabile da sostituire è la prima.

Una grammatica si dice **ambigua** se esiste $x \in L(G)$ per la quale ci siano più derivazioni non equivalenti.

La grammatica G_1 descritta a inizio capitolo è, quindi, non ambigua. Consideriamo ora la seguente grammatica G_2: $E \to E + E \mid E \cdot E \mid (E) \mid a$. Si ha $L(G_2) = L(G_1)$ ma stavolta G_2 è ambigua, e lo si può dimostrare dando due diversi alberi di derivazione per la stessa stringa. Descriviamo la stringa $a + a \cdot a$:





Per chiarire meglio il concetto diamo anche le due derivazioni. Per l'albero a sinsitra: $E \Rightarrow E + E \Rightarrow a + E \Rightarrow a + E \cdot E \Rightarrow a + a \cdot E \Rightarrow a + a \cdot a$, mentre per quello a destra: $E \Rightarrow E \cdot E \Rightarrow E + E \cdot E \Rightarrow a + E \cdot E \Rightarrow a + a \cdot E \Rightarrow a + a \cdot a$.

N.B.: Esistono linguaggi context-free per i quali non è possibile scrivere una grammatica non ambigua che li generi, questi linguaggi vengono definiti **inerentemente ambigui**, cioè che possono essere generati solo da grammatiche ambigue.

Proprietà di chiusura dei linguaggi context-free

Proposizione 1 (chiusura rispetto l'unione): Se L_1 e L_2 sono linguaggi context-free, allora anche $L_1 \cup L_2$ lo è
Dimostrazione: Siano $G_1 = (V_1, \Sigma, R_1, S_1)$ e $G_2 = (V_2, \Sigma, R_2, S_2)$ grammatiche con $V_1 \cap V_2 \neq \emptyset$ che generano rispettivamente L_1 e L_2. Allora la grammatica $G = (V_1 \cup V_2 \cup \{S\}, \Sigma, R_1 \cup R_2 \cup \{S \to S_1 | S_2\}, S)$, ed è evidente che questa grammatica genera $L_1 \cup L_2$, come volevasi dimostrare.

Proposizione 2 (chiusura rispetto la concatenazione): Se L_1, L_2 sono context-free, lo è anche $L_1 L_2$.
Dimostrazione: Consideriamo $G_1 = (V_1, \Sigma, R_1, S_1)$ e $G_2 = (V_2, \Sigma, R_2, S_2)$ grammatiche con $V_1 \cap V_2 \neq \emptyset$, tali che $L_1 = L(G_1)$ e $L_2 = L(G_2)$. Allora $G = (V_1 \cup V_2 \cup \{S\}, \Sigma, R_1 \cup R_2 \cup \{S \to S_1 S_2\}, S)$ genera $L_1 \cup L_2$.

Ricordiamo che se $L \subseteq \Sigma^*$, $L^* = L^* L \cup \{\varepsilon\}$.
Proposizione 3 (chiusura rispetto l'operatore stella): Se L_1 è un linguaggio context-free, lo è anche L^*.
Dimostrazione: Sia $G_1 = (V_1, \Sigma, R_1, S_1)$ la grammatica che genera L_1, e consideriamo la seguente grammatica $G = (V_1 \cup \{S\}, \Sigma, R_1 \cup \{S \to SS_1 | \varepsilon\}, S)$, questa è tale che $L^* = L(G)$, C.V.D.

Proposizione 4 (linguaggi finiti sono context-free): Se $L \subseteq \Sigma^*$ è finito, allora è context-free.
Dimostrazione: Sia $L = \{w_1, \dots, w_n\}$. La grammatica le cui regole sono $S \to w_1 | \dots | w_n$ (quindi ci sono tante regole quante parole del linguaggio L) genera evidentemente L, come volevasi dimostrare.

Da queste proposizioni segue che la famiglia dei linguaggi context-free contiene tutti i linguaggi regolari.
Corollario: Se L è regolare, è context-free.
Dimostrazione: Dal teorema di Kleene, L si ottiene da linguaggi finiti (e quindi context-free) attraverso un numero finito di operazioni di $\cup$, $\cdot$ e $*$. La tesi segue dalle proposizioni di chiusura sopracitate, C.V.D.

Esempio: Sia $L = \{a^i b^j c^k | i, j, k \geq 0 \wedge (i = j \wedge i = k)\}$. Dimostriamo che L è context-free scrivendo una grammatica che lo genera: si ha $L = L_1 \cup L_2$, con $L_1 = \{a^n b^n c^k | n, k \geq 0\}$ e $L_2 = \{a^n b^j c^n | j, n \geq 0\}$ e quindi possiamo dimostrare che L è context-free scrivendo delle grammatiche che generano L_1 e L_2 rispettivamente; la nostra grammatica sarà quella che genererà la loro unione. Definiamo quindi $S \to S_1 | S_2$ con $S_1 \to XC$, dove $X \to aXb | \varepsilon, C \to cC | \varepsilon$, e $S_2 \to aS_2 c | B$, dove $B \to bB | \varepsilon$. Si noti, inoltre, che questa grammatica che genera L, essendo definita come unione di due linguaggi non disgiunti, è ambigua. Infatti, sia ad esempio la parola abc (basterebbe anche prendere solo la parola vuota), si ha $S \Rightarrow S_1 \Rightarrow XC \Rightarrow AXbC \Rightarrow abC \Rightarrow abcC \Rightarrow abc$ ed anche $S \Rightarrow S_2 \Rightarrow aS_2 c \Rightarrow aBc \Rightarrow abBc \Rightarrow abc$.
Esercizio: disegnare gli alberi di derivazione per $abc \in L$.

Esercizi: Scrivere le regole dei CFG che generano i seguenti linguaggi. In tutti, $\Sigma = \{0,1\}$

a) $\{w | w \text{ contiene al massimo tre } 1\}$
b) $\{w | w \text{ inizia e finisce con lo stesso simbolo}\}$
c) $\{w | \text{la lunghezza di } w \text{ è dispari}\}$
d) $\{w | \text{la lunghezza di } w \text{ è dispari e il simbolo centrale è } 0\}$
e) $\{w | w = \tilde{w}, \text{ ovvero } w \text{ è un palindromo}\}$
f) L'insieme vuoto

Soluzioni: Riportiamo una possibile soluzione degli esercizi più rappresentativi

b) $S \to 0 \mid 1 \mid 0X0 \mid 1X1$
   $X \to \varepsilon \mid 0X \mid 1X$ (produce la parola vuota o una parola che inizia con 0 + qualcosa o 1 + qualcosa)
c) $S \to 0 \mid 1 \mid OP \mid 1P$
   $P \to \varepsilon \mid 00P \mid 01P \mid 10P \mid 11P$ (una parola pari si produce con due simboli più una parola pari)
d) $S \to 0 \mid 0S0 \mid 0S1 \mid 1S0 \mid 1S1$ (NB: "aggiungendo pure "|1" si ha un'altra possibile soluzione per c)
e) $S \to \varepsilon \mid 0 \mid 1 \mid 0S0 \mid 1S1$

Grammatiche Regolari

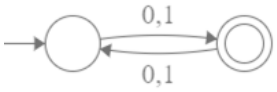
Sia $\mathcal{M} = (q, \Sigma, \delta, q_1, F)$ un DFA e costruiamo a partire da $\mathcal{M}$ una CFG G tale che $L(G) = L(\mathcal{M})$. Sia V un insieme di variabili della stessa cardinalità di Q; chiamiamo $X_i \in V$ la variabile corrispondente a $q_i \in Q$.

Definiamo la nostra grammatica $G = (V, \Sigma, R, X_1)$, dove R contiene le regole seguenti:

- Se $\delta(q_i, a) = q_j$, allora $X_i \to aX_j$ (praticamente scriviamo una regola per ogni freccia dell'automa)
- Se $q_i \in F$, allora $X_i \to \varepsilon$ (abbiamo così tradotto anche gli stati terminali)
  (In alternativa: se $q_i \in F \setminus \{q_1\}$, e $\delta(q_k, a) = q_i$ si può scrivere semplicemente $X_k \to a$)

Allora $L(G) = L(\mathcal{M})$. In generale una CFG è detta **regolare** se tutte le regole hanno la forma $X \to aY$ oppure $X \to a$ oppure $X \to \varepsilon$ (il viceversa non è vero).

Esempio: si prenda l'automa dell'esercizio c descritto negli esercizi del capitolo precedente:



corrisponde alla grammatica regolare $\begin{matrix} S \to 0X \mid 1X \\ X \to 0S \mid 1S \mid \varepsilon \end{matrix}$ o $\begin{matrix} S \to 0X \mid 1X \mid 0 \mid 1 \\ X \to 0S \mid 1S \mid \varepsilon \end{matrix}$

19. Automi pushdown (a pila) e loro corrispondenza con le CFG

Automi pushdown (a pila)

Aggiungono alle normali operazioni previste, una forma di I/O su una memoria (pila) non limitata, secondo il paradigma LIFO (Last In, First Out).

Ad ogni passo, le operazioni possibili (tutte facoltative) saranno:

- Lettura della prossima lettera della parola di input
- Estrazione del primo simbolo dalla pila (POP)
- Scrittura di un simbolo in cima alla pila (PUSH)

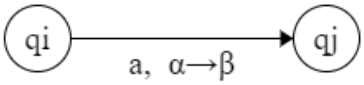
Dopodiché (con una o più operazioni) si passa agli stati successivi.

Formalmente, un PDA (automa pushdown) è una 6-upla: $(Q, \Sigma, \Gamma, \delta, q_1, F)$ con:

- $\boldsymbol{Q}$ insieme di **stati**
- $\boldsymbol{\Sigma}$ **alfabeto di terminali**
- $\boldsymbol{\Gamma}$ **alfabeto della pila**
- $\boldsymbol{\delta}$: $Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \to \mathcal{P}\big(Q \times (\Gamma \cup \{\varepsilon\})\big)$ **funzione di transizione** (parte da terne e restituisce un insieme di coppie stato, simbolo dell'alfabeto della pila Γ o nessun simbolo).
- $\boldsymbol{q_1}$ **stato iniziale**
- $\boldsymbol{F} \subseteq Q$ **stati di accettazione**

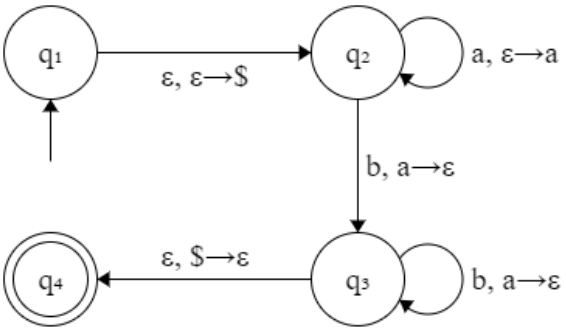
Una parola $w \in \Sigma^*$ è accettata da $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_1, F)$ se $w = w_1 \dots w_n, n \geq 0$ e $w_i \in \Sigma \cup \{\varepsilon\}$ per $i = 1, \dots, n$. Inoltre, $\exists q_2, \dots, q_{n+1} \in Q, s_1 = \varepsilon, s_2, \dots, s_n \in \Gamma^*$ tali che per $i = 1, \dots, n$, $(q_{i+1}, \beta_i) \in \delta(q_i, w_I, \alpha_i)$, e le due parole $s_i = \alpha_i t_i$ e $s_{i+1} = \beta_i t_i$ per $t_i \in \Gamma^*, \alpha_i \in \Gamma \cup \{\varepsilon\}, q_{n+1} \in F$ (q_{n+1} deve essere di accettazione). Praticamente, le parole $s_i \in \Gamma^*$ rappresentano tutto ciò che è scritto nella pila a un dato istante.

Nota: $(q_j, \beta) \in \delta(q_i, a, \alpha)$ si rappresenta nel diagramma con



Esempi (mostrano tutta la potenza del non determinismo di questi automi)

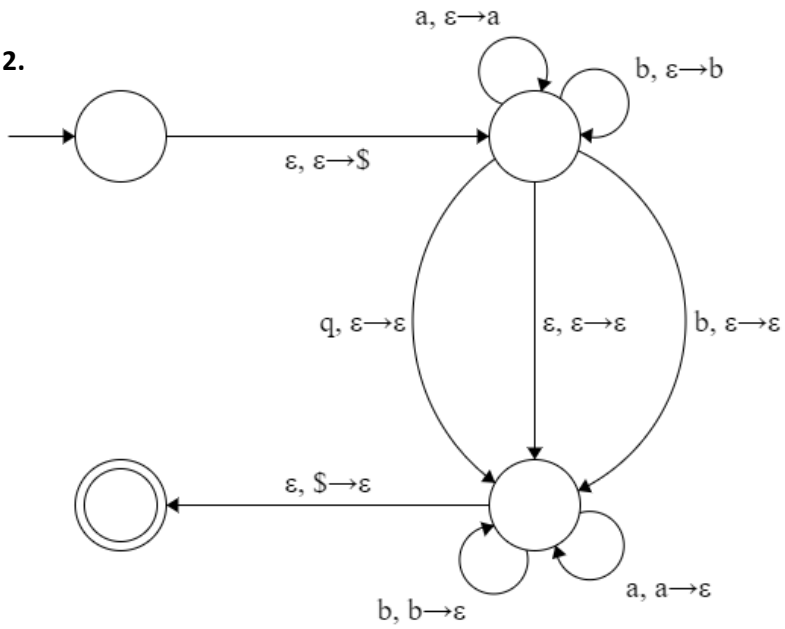
1.



Il carattere speciale $ ci serve a capire quando la pila sarà svuotata. Il diagramma accetta il seguente linguaggio:

$$L(\mathcal{M}_1) = \{a^n b^n | n > 0\}$$

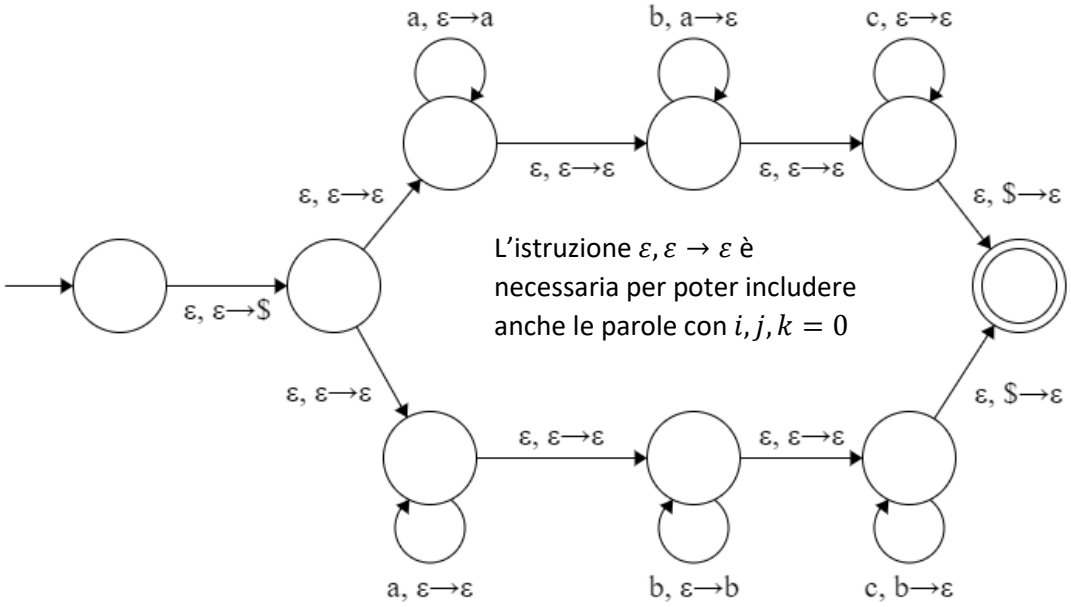
2.



Anche se più complesso, questo automa accetta la parola solo allo svuotamento della pila (come il precedente). Il linguaggio accettato è il seguente:

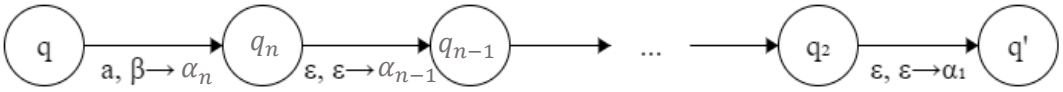
$$L(\mathcal{M}_2) = \{w \in \{a, b\}^* | w = \tilde{w}\}$$

3. Dato un CFG che generi $L = \{a^i b^j c^k | i = j \text{ oppure } j = k,\ i,j,k \geq 0\}$ con le seguenti regole:
$S \to XC|AY \qquad X \to aXb|\varepsilon \qquad Y \to bYc|\varepsilon \qquad A \to aA|\varepsilon \qquad C \to cC|\varepsilon$
Il diagramma di un PDA che accetti L è il seguente

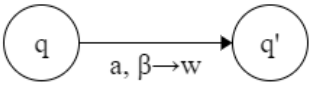


Corrispondenza con le grammatiche context-free dei PDA

Notazione: Se $a \in \Sigma \cup \{\varepsilon\}$, $\beta \in \Gamma \cup \{\varepsilon\}$, e $w = \alpha_1 \dots \alpha_n \in \Gamma^*$ ($\alpha_i \in \Gamma$ per $i = 1, \dots, n$) scriviamo (praticamente definiamo δ^*) $(q', w) \in \delta^*(q, a, \beta)$ se $\exists q_1, \dots, q_n \in Q : (q_n, \alpha_n) \in \delta(q, a, \beta)$ e $\delta(q_n, \varepsilon, \varepsilon) = \{(q_{n-1}, \alpha_{n-1})\}, \dots, \delta(q_{i+1}, \varepsilon, \varepsilon) = \{(q_i, \alpha_i)\}$ per $i = 1, \dots, n$ e lo stato q' è proprio q_1. Nel diagramma ciò corrisponde a

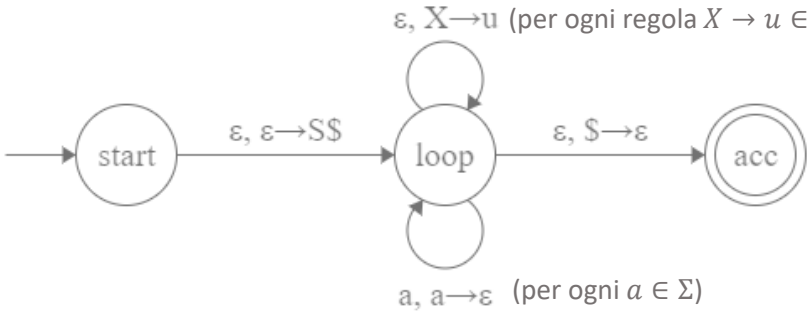


che verrà sintetizzato semplicemente con



Teorema: $L \subseteq \Sigma^*$ context-free $\Leftrightarrow$ L accettato da un PDA.

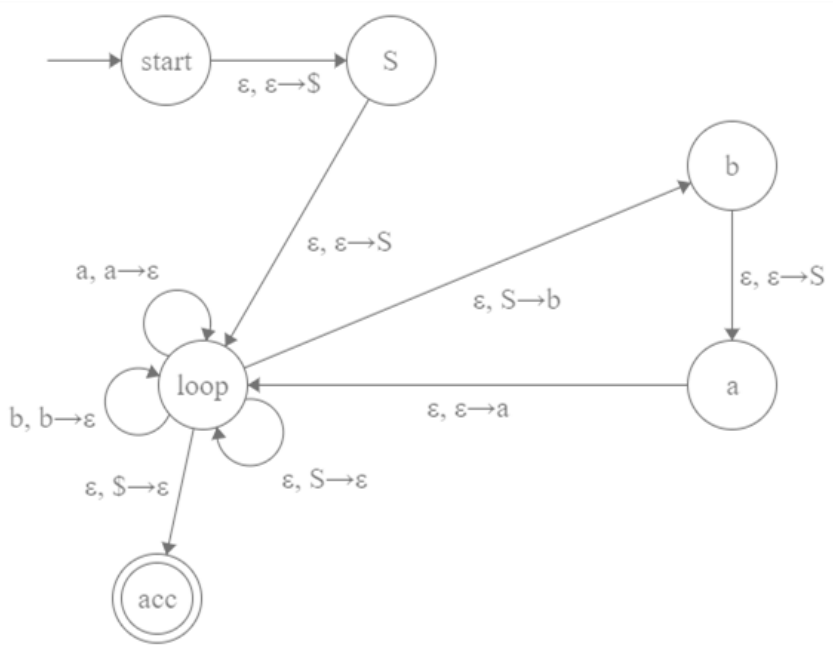
Dimostrazione $\Rightarrow$ (se L è CF, è accettato da un PDA): Sia $G = (V, \Sigma, R, S)$ una CFG tale che $L = L(G)$. Disegniamo il diagramma di un PDA, semplificato con la notazione sopracitata:



Tale automa accetta esattamente L, come volevasi dimostrare (l'altra implicazione non verrà dimostrata).

Poiché un NDFA non è altro che un PDA in cui la pila non viene utilizzata e in ogni transizione viene letta una lettera dell'input, abbiamo una terza dimostrazione del fatto che **tutti i linguaggi regolari sono CF**

Esempio: Consideriamo la grammatica G_1 con regole $S \to aSb|\varepsilon$ (disegnando anche gli stati intermedi implicitati nella dimostrazione del teorema precedente):



20. Pumping lemma e gerarchia Chomsky-Schützenberger

Esprime una condizione necessaria ma non sufficiente affinché un linguaggio sia context-free.

Forma normale di Chomsky

Una CFG si dice in **forma normale di Chomsky** se tutte le sue regole hanno una tra le seguenti forme:

$$\begin{cases} X \to YZ, & \text{con } Y, Z \in V \setminus \{S\} \\ X \to a, & \text{con } a \in \Sigma \\ S \to \varepsilon, & \text{altrimenti} \end{cases}$$

Teorema: Per ogni linguaggio CF L, esiste una CFG G in forma normale di Chomsky tale che $L(G) = L$.

La forma normale di Chomsky è utile tra l'altro per velocizzare algoritmi di pressing (che determinano se una data stringa appartiene o meno al linguaggio generato da una grammatica context-free).
Dato che per una grammatica in forma normale tutte le regole hanno parte destre di lunghezza ≤ 2, gli alberi di derivazione corrispondenti sono binari.

Per convertire una CFG si esegue un algoritmo a 5 passi, descriviamo solo il passo della conversione che permette di arrivare a regole con lunghezza della parte destra ≤ 2 (ovvero il passo 3):

- Supponiamo che R (insieme di regole della grammatica) contenga la seguente regola
  $X \to u_1 u_2 \dots u_k$, con $u_i \in (V \cup \Sigma)$ per $i = 1, \dots, k$, e $k > 2$
  Allora introduciamo nuove variabili: $X_1, \dots, X_{k-2}$. Dopodiché sostituiamo la regola precedente con:
  $$\begin{aligned} X &\to u_1 X_1 \\ X_1 &\to u_2 X_2 \\ &\vdots \\ X_{k-3} &\to u_{k-2} X_{k-2} \\ X_{k-2} &\to u_{k-1} u_k \end{aligned}$$

Esempio di conversione: $S \to aSb|\varepsilon$

1) $S \to X$
   $X \to aXb|\varepsilon$

2) $S \to X$
$X \to AXB|\varepsilon$
$A \to a$
$B \to b$

3) $S \to X$
$X \to AY|\varepsilon$
$Y \to XB$
$A \to a$
$B \to b$

4) $S \to X|\varepsilon$
$X \to AY$
$Y \to XB|B$
$A \to a$
$B \to b$

5) $S \to AY|\varepsilon$
$X \to AY$
$Y \to XB|b$
$A \to a$
$B \to b$

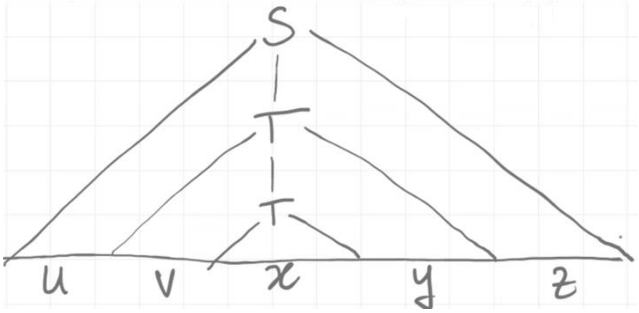
All'esame non verrà mai richiesto di convertire una grammatica in forma di Chomsky, al massimo verrà richiesto di dire se una data grammatica è in forma di Chomsky.

Pumping lemma per i linguaggi context-free

Teorema: Se $L \subseteq \Sigma^*$ è context-free, $\exists p \geq 0 : \forall w \in L, |w| \geq p \Rightarrow \exists u, v, x, y, z \in \Sigma^* : w = uvxyz$ e

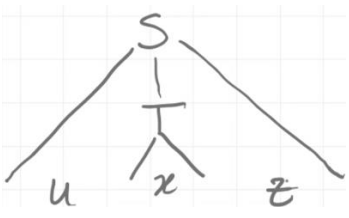
1) $\forall i \geq 0, uv^i xy^i z \in L$
2) $|vy| > 0$ (le parole v e y non possono essere entrambe vuote)
3) $|vxy| \leq p$ (la parte centrale della decomposizione non può essere maggiore di p)

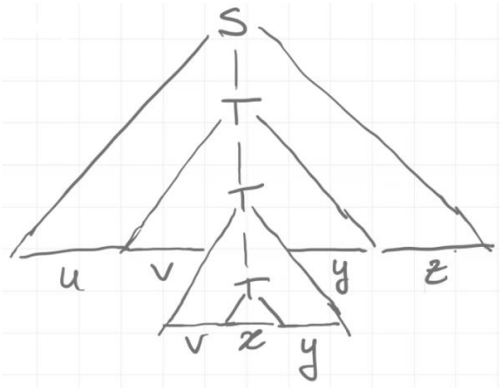
Dimostrazione: Sia $G = (V, \Sigma, R, S)$ una CFG che genera L, in forma normale di Chomsky (poiché ci interessa avere regole con lunghezza della parte destra ≤ 2). Poniamo $p = 2^{card(V)+1}$, cioè: se $V = \{S = X_1, \dots, X_k\}$, allora pongo 2^{k+1}. Un albero di derivazione di profondità 1 (tipi possibili: ⩚ |) corrisponde a parole di lunghezza al massimo 2. In generale, alberi di profondità h corrispondono a parole di lunghezza $\leq 2^h$. Quindi se $|w| \geq p = 2^{k+1}$, un albero di derivazione per w ha profondità almeno $k+1$; di conseguenza, in tale albero, un cammino massimale dalla radice a una foglia contiene $k+1$ nodi interni. Ma essendo per ipotesi solo k variabili, per il principio della piccionaia in tale percorso almeno una tra le variabili si ripete almeno due volte (altrimenti non potrei disporre k variabili in $k+1$ posti). Lo stesso ragionamento vale se anziché partire dalla radice partiamo da $k+1$ nodi al di sopra di una foglia. Dunque, la situazione a cui siamo arrivati può rappresentarsi graficamente con il seguente albero di derivazione:



Dove x è la parola che si legge da sinistra a destra delle foglie del sottoalbero radicato nella seconda T, vxy la parola radicata nell'albero della prima T e con $uvxyz$ abbiamo praticamente la decomposizione della nostra parola.

Dimostriamo ora il punto due: se per assurdo $v = y = \varepsilon$, allora $w = uxz$. Supponiamo che l'albero precedente abbia numero minimo di nodi. Ma dato che $w = uxz$, w avrà anche l'albero di derivazione rappresentato a destra; in cui abbiamo sostituito al primo sottoalbero generato da T il secondo. Questo ci porta ad ottenere la parola $uxz = w$ ma con un numero minore di nodi, ma ciò è assurdo poiché abbiamo supposto quello di prima minimale (e quello nuovo non può ovviamente avere meno nodi).





Dimostrazione del punto 1: Quanto visto al punto 2 mostra che $uxz = uv^0xy^0z \in L$. Sostituendo invece il sottoalbero generato dalla seconda occorrenza di T con il primo, otteniamo il sottoalbero in figura a sinistra; il che mostra che $uv^2xy^2z \in L$. Iterando questo procedimento avremo il caso $uv^ixy^iz \in L$ per $i \geq 0$.
Dimostrazione del punto 3: Da quanto detto in precedenza, l'albero corrispondente alla prima occorrenza di T ha profondità $\leq k + 1$. Dunque, la parola corrispondente, vxy, ha lunghezza $\leq 2^{k+1} = p$, come volevasi dimostrare.

Conseguenze del pumping lemma

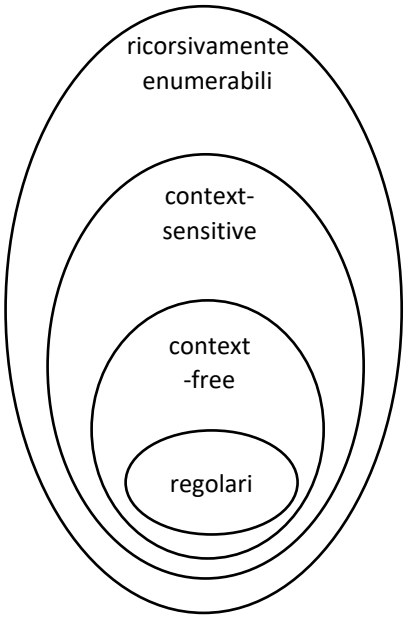
Esempio: $L = \{a^n b^n c^n | n \geq 0\}$ non è CF. Infatti, supponiamo per assurdo che lo sia, e sia p la lunghezza di pumping. Consideriamo $a^p b^p c^p \in L$ e sia $a^p b^p c^p = uvxyz$ (la decomponiamo in 5 parti), con $|vy| > 0$. Se v e y contengono lettere di un solo tipo, ad esempio, se $v \in \{a\}^*$ e $y \in \{c\}^*$, chiaramente uv^2xy^2z non ha più lo stesso numero di a, b e c e dunque non appartiene a L. Altrimenti, o v oppure y (o entrambe) contengono lettere di più tipi, ad esempio $v = a^j b^k$, allora $uv^2xy^2z \notin \langle a^* b^* c^* \rangle$ e dunque non appartiene neanche a L. Ciò mostra che il pumping lemma non sarebbe valido; l'assurdo segue dall'aver supposto L context-free, abbiamo così dimostrato che L non è context-free.

Gerarchia di Chomsky-Schützenberger

È una classificazione di linguaggi che va dal caso più semplice (quelli regolari) a quello più complesso (i ricorsivamente enumerabili). La figura a destra mostra come sono raggruppati i linguaggi visti sino ad ora, con context-sensitive (non trattata) si intende la grammatica sensibile al contesto.

Linguaggi	**Grammatiche**	**Automi**
Regolari	$X \to aY$ Regolari: $X \to a$ $X \to \varepsilon$	DFA NDFA
Context-free	CFG: $X \to w$ con $w \in (V \cup \Sigma)^*$	PDA
Context-sensitive	CSG: $\alpha X \beta \to \alpha w \beta$ con $\alpha, \beta \in (V \cup \Sigma)^*$, $w \in (V \cup \Sigma)^* \setminus \{\varepsilon\}$	Macchine di Turing limitate linearmente
Ricorsivamente enumerabili	Senza restrizioni: $u \to w$ con $u, w \in (V \cup \Sigma)$	Macchine di Turing

Tabella riepilogativa dei linguaggi



N.B.: La famiglia dei linguaggi CF **non** è chiusa rispetto all'intersezione.
Infatti, ad esempio, i linguaggi $L_1 = \{a^i b^j c^j | i, j \geq 0\}$ e $L_2 = \{a^i b^i c^j | i, j \geq 0\}$ sono CF, ma $L_1 \cap L_2 = \{a^n b^n c^n | n \geq 0\}$ non lo è. Le leggi di De Morgan implicano allora che la famiglia deli linguaggi CF non è chiusa neanche rispetto al complemento.