Fondamenti dell'informatica
UNIVR - Dipartimento di Informatica

"*Numquam moriuntur vincula*"

"*Natura rerum intellegénda est*"

Amos Lo Verde

15 dicembre 2024

Indice

Prefazione

Questa dispensa è stata scritta mettendo assieme le nozioni prese dalle: lezioni seguite in presenza (A.A. 2022/2023), videolezioni caricate sulla piattaforma Moodle e dal testo di riferimento consigliato per il corso.
L'utilizzo di questo materiale non deve sostituire la frequentazione del corso, ma solo supportarlo a fine didattico.

È possibile condividere gratuitamente questa dispensa. Inoltre si ricorda che tutti i materiali sono sempre disponibili al canale Telegram `https://t.me/univrinfo`.

Crediti

Professore del corso (A.A. 2022/2023): Roberto Giacobazzi.

Testo di riferimento consigliato: *Fondamenti dell'informatica. Linguaggi formali, calcolabilità e complessità* (Agostino Dovier, Roberto Giacobazzi).

Notazione e concetti base

Le **funzioni** vengono rappresentate mediante il loro **grafico**.

Esempio

Il grafico della funzione che calcola numeri pari consiste in:

$$f = \{(0,0),(1,2),(2,4),(3,6),\dots\}$$

Questa funzione prende in *input* un valore x e restituisce $x \cdot 2$: $\lambda x.x \cdot 2$.

Un **algoritmo** è una **sequenza finita** di simboli (istruzioni).

Esempio

L'algoritmo Q della precedente funzione corrisponde a:

```
input(x);
x = x * 2;
output(x);
```

L'algoritmo è **intensionale**[1], mentre il grafico è **estensionale**[2]. Ci sono infinite varianti intensionali, ma che corrispondono a una singola versione estensionale.

Si può dire che l'algoritmo Q implementa f. In generale Q appartiene a un certo **linguaggio** $\mathscr{L}$ e la sua esecuzione avente in *input* 'i' restituisce in *output* 'o' se e solo se la coppia (*input*, *output*) appartiene al grafico di f:

$$Q \in \mathscr{L} \land [\![Q]\!]i = o \iff (i,o) \in f$$

$\mathscr{L}$ è l'**insieme di tutti i programmi** che possono essere scritti con quel linguaggio. Se $Q \in \mathscr{L}$, allora l'algoritmo Q è **corretto**.

[1] Il modo in cui viene scritto.

[2] Descrizione nei dettagli di cosa prende in *input* e cosa restituisce in *output*.

In questa struttura è importante sapere quante sono le funzioni che si possono implementare in un dato linguaggio.

Si definisce la classe di tutte le funzioni del linguaggio $\mathscr{L}$ come:

$$F_{\mathscr{L}} = \{f : \mathbb{N} \to \mathbb{N} \mid f = [\![P]\!],\ P \in \mathscr{L}\}$$

Questo è lo spazio di tutte le funzioni calcolabili, cioè le funzioni implementabili nel linguaggio $\mathscr{L}$ generico.

Osservazione

La dimensione delle funzioni in $\mathscr{L}$ è uguale alla dimensione di $\mathscr{L}$:

$$|F_{\mathscr{L}}| = |\mathscr{L}| = |\mathbb{N}| = \omega$$

Pertanto ci sono tante funzioni programmabili in $\mathscr{L}$ quanti i programmi che si possono scrivere. Essendo che i programmi sono descritti da sequenze finite di istruzioni, il numero di quest'ultimi è associabile a un numero naturale. ω è un limite ordinale dei numeri naturali.

Un programma prende un dato e lo trasforma in un altro dato:

$$\mathbb{N} \to \mathbb{N}$$

Qui dentro ci stanno i problemi estensionalmente definiti.

1.1 Cardinalità di Insiemi

▷ Poiché i problemi sono funzioni, in che relazione stanno con lo spazio delle funzioni che si possono calcolare in un linguaggio?
Dalla precedente osservazione è noto che la cardinalità dell'insieme $F_{\mathscr{L}}$ è ω. Per determinare la cardinalità dell'insieme di tutte le funzioni $\mathbb{N} \to \mathbb{N}$ si sfrutta il seguente teorema.

Teorema 1.1.1: Georg Cantor (1891)

La cardinalità dei numeri $\mathbb{N}$ (programmi) è strettamente minore della cardinalità $\mathbb{N} \to \mathbb{N}$:

$$|\mathbb{N}| < |\mathbb{N} \to \mathbb{N}|$$

Dato che la cardinalità di $\mathbb{N}$ è la cardinalità dell'insieme rappresentabile attraverso le funzioni, allora si scopre che esistono molti più problemi (molte più funzioni) di quelli che si possono rappresentare in un qualsiasi linguaggio di programmazione, nell'ipotesi che i programmi abbiano dimensione finita.

Dimostrazione

Si suppone per assurdo che $|\mathbb{N} \to \mathbb{N}| = |\mathbb{N}|$, ossia ci sono tante funzioni quanti i numeri naturali. Se vale questa uguaglianza, allora implica l'esistenza di un'enumerazione per tutte le funzioni $|\mathbb{N}| \to |\mathbb{N}| : f_0, f_1, f_2, \ldots, f_x, \ldots$:

$$\forall f \in \mathbb{N} \to \mathbb{N}, \quad \exists i \in \mathbb{N} : \ f = f_i$$

A questo punto si definisce una funzione g, che prende in *input* un numero x, e restituisce il valore della funzione x-esima sommato più 1:

$$g(x) = f_x(x) + 1$$

- L'oggetto x, di $g(x)$, appartiene a $\mathbb{N}$.
- La funzione f_x sta in $\mathbb{N} \to \mathbb{N}$, a cui si passa un elemento che sta in $\mathbb{N}$, perciò il risultato è $\mathbb{N}$. A questo numero viene sommato 1 e si ottiene nuovamente un numero in $\mathbb{N}$.

Di conseguenza $g(x) \in \mathbb{N} \to \mathbb{N}$. Tuttavia questa funzione porta a un assurdo: poiché $g(x) = f_x(x) + 1 \in \mathbb{N} \to \mathbb{N}$, allora prima o poi deve comparire all'interno della enumerazione definita in precedenza $(f_0, f_1, f_2, \ldots, f_x, \ldots)$, cioè deve avere un indice:

$$\exists n_0 \in \mathbb{N} : \ g = f_{n_0}$$

Se esiste un indice n_0 per la funzione g, allora si passa quell'indice:

$$g(n_0) = f_{n_0}(n_0)$$

Ma per definizione $g(x) = g(n_0) = f_{n_0}(n_0) + 1$, perciò si verifica un assurdo, perché:

$$f_{n_0}(n_0) = g(n_0) = f_{n_0}(n_0) + 1$$

Se ciò fosse vero significherebbe che un numero è uguale al suo successore. Pertanto non si possono enumerare le funzioni $|\mathbb{N}| \neq |\mathbb{N} \to |\mathbb{N}|$.

Osservazione

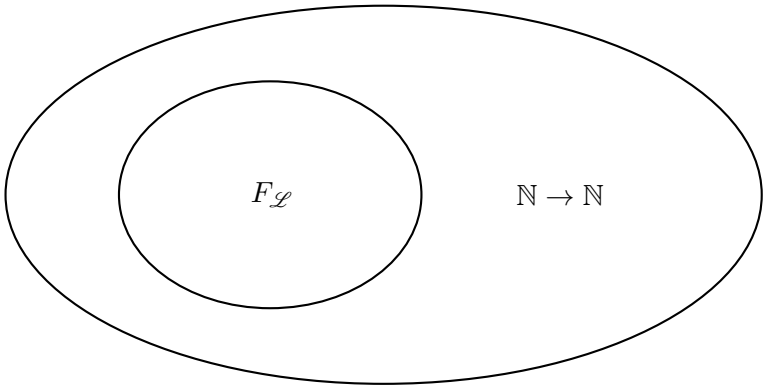
Questa dimostrazione è valida soltanto per problemi totali, ma non per quelli parziali. Poiché se le funzioni non terminassero, cioè non arrivano mai alla conclusione, allora si avrebbe:

$$\underbrace{f_{\overline{n}}(\overline{n})}_{\infty} = \underbrace{f_{\overline{n}}(\overline{n}) + 1}_{\infty}$$

La conseguenza di questo ragionamento comporta che:

$$\left.\begin{array}{l} |F_{\mathscr{L}}| = |\mathbb{N}| = \omega \\ |\mathbb{N} \to \mathbb{N}| > |\mathbb{N}| = \omega \end{array}\right\} |F_{\mathscr{L}}| < |\mathbb{N} \to \mathbb{N}|$$

La relazione passa da $|\mathbb{N}| \leq |\mathbb{N} \to \mathbb{N}|$ a $|\mathbb{N}| < |\mathbb{N} \to \mathbb{N}| = |\mathbb{R}|$, assumendo l'ipotesi del continuo.



Questa relazione afferma che i problemi in $\mathbb{N} \to \mathbb{N}$ non sono risolvibili attraverso un programma che si può scrivere con un linguaggio.

Osservazione

I problemi sono tanti quanti i numeri reali, mentre le soluzioni algoritmiche ai problemi sono tante quanti i numeri naturali.

In generale la cardinalità di $\mathbb{N} \to \mathbb{N}$ è tanta quanta la cardinalità delle parti[3] di $\mathbb{N}$:

$$|\mathbb{N} \to \mathbb{N}| = |2^{\mathbb{N}}|$$

Questa relazione è ottenuta dalla seguente considerazione: la cardinalità delle parti di $\mathbb{N}$, cioè dei sottoinsiemi dei numeri naturali, è uguale alla cardinalità delle funzioni da $\mathbb{N}$ a $\{0, 1\}$:

$$|2^{\mathbb{N}}| = |\mathbb{N} \to \{0, 1\}|$$
$$2^{\mathbb{N}} = \{s \mid s \subseteq \mathbb{N}\}$$

Per verificarlo si considera $s \subseteq \mathbb{N}$ che può essere rappresentato dalla funzione caratteristica f_s, sviluppata nella seguente maniera:

$$s \subseteq \mathbb{N} \rightsquigarrow f_s(x) = \begin{cases} 1 & x \in s \\ 0 & x \notin s \end{cases} \to \text{funzione caratteristica di } s$$

[3] I sottoinsiemi dei numeri naturali.

Quindi a ogni sottoinsieme dei numeri naturali si può associare una funzione e viceversa, data una funzione $\mathbb{N} \to \{0,1\}$, si può associare un insieme:

$$f : \mathbb{N} \to \{0,1\} \rightsquigarrow \{x \mid f(x) = 1\} \subseteq \mathbb{N}$$

Questi problemi sono detti **problemi decisionali**.

1.2 Problemi decisionali

▷ Quanti sono i **problemi decisionali**?
Le funzioni da $\mathbb{N} \to \{0,1\}$ sono tante quante le funzioni da $\mathbb{N} \to \mathbb{N}$:

$$|\mathbb{N} \to \{0,1\}| = |\mathbb{N} \to \mathbb{N}|$$

Dimostrazione

Si verificano le due inclusioni per $|\mathbb{N} \to \{0,1\}| \begin{matrix}\subseteq \\ \supseteq\end{matrix} |\mathbb{N} \to \mathbb{N}|$:

- **Prima inclusione** $\subseteq$: tutte le funzioni da $\mathbb{N} \to \mathbb{N}$ sono maggiori o uguali alle funzioni da $\mathbb{N} \to \{0,1\}$:

$$|\mathbb{N} \to \{0,1\}| \leq |\mathbb{N} \to \mathbb{N}|$$

- **Seconda inclusione** $\supseteq$: le funzioni da $\mathbb{N} \to \{0,1\}$ sono maggiori o uguali alle funzioni da $\mathbb{N} \to \mathbb{N}$:

$$|\mathbb{N} \to \{0,1\}| \geq |\mathbb{N} \to \mathbb{N}|$$

  Infatti sapendo che l'espressione $|\mathbb{N} \times \mathbb{N}| = |\mathbb{N}|$ è vera (non cambiano gli ordini di infinito) si prende una funzione $f : \mathbb{N} \to \mathbb{N}$, alla quale è associato un insieme $s_f = \{(i,o) \mid f(i) = 0\} \subseteq \mathbb{N} \times \mathbb{N}$.
  Dunque tra tutte le coppie, la coppia (*input*, *output*) dà il grafico della funzione f; per ogni funzione è associato un sottoinsieme $\mathbb{N} \to \mathbb{N}$. Allora la funzione caratteristica associabile a s_f è:

$$\underbrace{C_{s_f}(i,0)}_{\in \mathbb{N}\times\mathbb{N}\to\{0,1\}} = \begin{cases} 1 & \text{se } (i,o) \in s_f \\ 0 & \text{altrimenti} \end{cases}$$

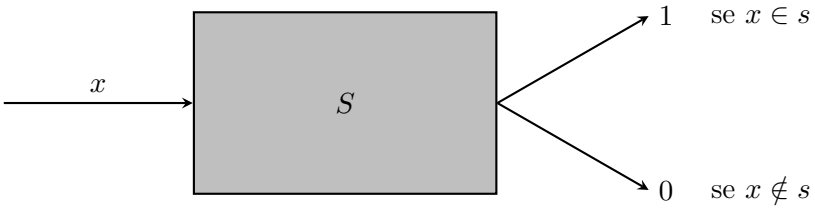
  Ne consegue che la cardinalità di tutte le funzioni sia uguale alla cardinalità delle funzioni $\mathbb{N} \times \mathbb{N} \to \{0,1\}$:

$$\begin{aligned} |\mathbb{N}| < |\mathbb{N} \to \mathbb{N}| &= |\mathbb{N} \times \mathbb{N} \to \{0,1\}| \\ &= |\mathbb{N} \to \{0,1\}| = |2^{\mathbb{N}}| = |\mathbb{R}| \end{aligned}$$

Gli algoritmi in $\mathbb{N}$ sono sempre meno del:

- numero di problemi totali esistenti;
- numero di problemi decisionali;
- sottoinsiemi di $\mathbb{N} = 2^{\mathbb{N}}$.

Tutto ciò che è stato analizzato fin'ora ha il seguente significato: se si suppone di avere una "scatola", che restituisce i valori 1 o 0 a seconda che l'*input* x appartenga a un insieme s oppure no, allora l'ipotesi di programmare una macchina del genere ha probabilità nulla se si considera $s \subseteq \mathbb{N}$.



Quindi solo alcuni sottoinsiemi di $\mathbb{N}$ sono detti **decidibili**, ovvero quelli per cui esiste una macchina che afferma "sì" oppure "no" se l'*input* appartiene o meno all'insieme.

Automi a stati finiti

2.1 Linguaggi regolari

I **linguaggi regolari** rappresentano la famiglia dei modelli di calcolo più semplice; corrispondono al processore di una calcolatore. Nel dettaglio il processore è un **automa a stati finiti**.

I problemi relativi a questi linguaggi sono **decisionali**, ossia se una sequenza di simboli (stringa) appartiene a un linguaggio regolare, allora è risolvibile da un algoritmo e implementabile mediante una macchina a stati finiti.

2.1.1 Alfabeto, simboli e stringhe

Un **simbolo** è un'entità primitiva astratta non definita formalmente, come lettere e caratteri numerici. L'**alfabeto** è un insieme di simboli denotato con Σ e avente le seguenti caratteristiche:

- La cardinalità di Σ è minore di ω:

  $$|\Sigma| < \omega$$

  Quindi Σ è un **insieme finito**.

- L'insieme $\Sigma \neq \emptyset$. Di conseguenza è presente almeno un simbolo.

L'**insieme delle stringhe** sull'alfabeto Σ è rappresentato da Σ^*: sono tutte le sequenze di simboli che appartengono all'alfabeto Σ.

Esempio

Se Σ è rappresentato dall'insieme binario $\{0, 1\}$, allora la stringa 101011 appartiene a $\{0, 1\}^*$:

$$\Sigma = \{0, 1\}, \quad 101011 \in \{0, 1\}^*$$

La cardinalità dell'insieme delle stringhe Σ^*, su qualunque alfabeto, è uguale alla cardinalità dei numeri naturali:

$$|\Sigma^*| = |\mathbb{N}| = \omega$$

Dato un alfabeto finito, le stringhe finite sono tanti quanti i numeri naturali. Dunque ci sono infinite stringhe finite di simboli.

Operazioni sulle stringhe

L'insieme delle stringhe è una struttura algebrica, per cui esistono diverse operazioni:

- **Concatenazione**: è un'operazione binaria indicata dal simbolo " $\cdot$ ". Prese due stringhe viene restituito in *output* una stringa data dalla concatenazione delle due stringhe:
$$\Sigma^* \times \Sigma^* \to \Sigma^*$$
  - Esiste una stringa particolare ε che appartiene a Σ^* detta **stringa vuota**. Questa ha la caratteristica di essere l'**unità** rispetto al prodotto dato dalla concatenazione.
$$\varepsilon \in \Sigma^*, \quad (\Sigma^*, \cdot) \text{ monoide}$$
L'insieme di tutte le stringhe con l'operazione binaria di concatenazione forma un **monoide**, cioè l'unità di stringa vuota.

    Esempio

    ε è una stringa vuota. Se si concatena una certa stringa non vuota σ a ε, allora il risultato è uguale a concatenare ε a σ, perciò uguale a σ:
$$\sigma \cdot \varepsilon = \varepsilon \cdot \sigma = \sigma$$
  - Per ogni coppia di stringhe, la loro concatenazione appartiene all'insieme delle stringhe:
$$\forall \sigma, \beta \in \Sigma^* : \sigma \cdot \beta \in \Sigma^*$$
  - Esistono due stringhe tali per cui la concatenazione non è commutativa:
$$\exists \alpha, \beta \in \Sigma^* : \alpha \cdot \beta \neq \beta \cdot \alpha$$
  - Per ogni coppia di stringhe, la lunghezza della stringa α concatenata a β è uguale alla lunghezza di α più la lunghezza di β:
$$\forall \alpha, \beta \in \Sigma^* : |\alpha \cdot \beta| = |\alpha| + |\beta|$$
- **Complemento**: si considera un linguaggio $\mathscr{L}$ contenuto in Σ^*. Allora si definisce $\overline{\mathscr{L}}$ corrispondente all'insieme di tutte le stringhe σ tale per cui σ è una stringa sullo stesso alfabeto e non appartiene a $\mathscr{L}$:
$$\overline{\mathscr{L}} = \{\sigma \mid \sigma \in \Sigma^*, \sigma \notin \mathscr{L}\} = \Sigma^* \setminus \mathscr{L}$$

- Si considerano due linguaggi $\mathscr{L}_1, \mathscr{L}_2$ sullo stesso alfabeto Σ. Allora si definisce $\mathscr{L}_1$ **concatenato** a $\mathscr{L}_2$ come l'insieme di tutte le stringhe α concatenate a β tali per cui α appartengano a $\mathscr{L}_1$ e β appartengano a $\mathscr{L}_2$:

$$\mathscr{L}_1 \cdot \mathscr{L}_2 = \{\alpha \cdot \beta \mid \alpha \in \mathscr{L}_1,\, \beta \in \mathscr{L}_2\}$$

  - È possibile iterare la concatenazione. Questa operazione è nota come **iterazione di Kleene** oppure *** di Kleene**. È definita nella seguente maniera: dato un linguaggio $\mathscr{L}$ contenuto in Σ^*, si ha che la concatenazione di $\mathscr{L}^*$ è uguale all'unione per tutti gli $n \geq 0$ delle potenze di $\mathscr{L}^n$:

$$\mathscr{L}^* = \bigcup_{n \geq 0}^{\infty} \mathscr{L}^n$$

  - Se la concatenazione "è come" il prodotto, allora si può immaginare la potenza di questo prodotto definita come:

$$\begin{cases} \mathscr{L}^0 = \{\varepsilon\} \\ \mathscr{L}^{n+1} = \mathscr{L}^n \cdot \mathscr{L} \end{cases}$$

    Se non si parte da $n \geq 0$, bensì da $n \geq 1$, allora si ottiene:

$$\mathscr{L}^+ = \bigcup_{n \geq 1}^{\infty} \mathscr{L}$$

Esempio

Viene dato l'alfabeto $\Sigma = \{a, b\}$ e il linguaggio $\mathscr{L} = \{a, b\}$.
$\mathscr{L}^*$ e $\mathscr{L}^+$ corrispondono a:

$$\mathscr{L}^+ = \{a, b\} \cup \{aa, bb, ab, ba\} \cup \dots$$
$$\mathscr{L}^* = \{\varepsilon\} \cup \{a, b\} \cup \{aa, bb, ab, ba\} \cup \dots$$

Quindi il linguaggio $\mathscr{L}$ è finito, le iterazioni $\mathscr{L}^*$ e $\mathscr{L}^+$ sono infinite e inoltre quest'ultima non parte dalla stringa vuota ε.

Esempio

Dato l'alfabeto $\Sigma = \{a, b\}$ e il linguaggio $\mathscr{L} = \{a, b\}$, allora $\mathscr{L}^*$ consiste in:

$$\mathscr{L}^* = \{ \underbrace{\varepsilon}_{\mathscr{L}^0}, \underbrace{a, b}_{\mathscr{L}^1}, \underbrace{aa, ab, ba, bb}_{\mathscr{L}^2}, \underbrace{\ldots}_{\mathscr{L}^*} \}$$

Le sequenze in Σ^* sono costruite con l'iteratore $\mathscr{L}^*$ quando $\mathscr{L}$ coincide con Σ.

- **Unione**: dati due linguaggi $\mathscr{L}_1$ e $\mathscr{L}_2$ nello stesso alfabeto, allora l'unione $\mathscr{L}_1 \cup \mathscr{L}_2$ sono tutte le stringhe σ tali per cui σ appartiene a $\mathscr{L}_1$ oppure σ appartiene a $\mathscr{L}_2$:

$$\mathscr{L}_1 \cup \mathscr{L}_2 = \{\sigma \mid \sigma \in \mathscr{L}_1 \vee \sigma \in \mathscr{L}_2\}$$

- **Intersezione**: dati due linguaggi $\mathscr{L}_1$ e $\mathscr{L}_2$ nello stesso alfabeto, allora l'intersezione $\mathscr{L}_1 \cap \mathscr{L}_2$ è uguale a tutti i σ tali che σ appartengano sia a $\mathscr{L}_1$ sia a $\mathscr{L}_2$:

$$\mathscr{L}_1 \cap \mathscr{L}_2 = \{\sigma \mid \sigma \in \mathscr{L}_1 \wedge \sigma \in \mathscr{L}_2\}$$

Linguaggio

A partire dagli insiemi di sequenze di simboli (stringhe) si formano i **linguaggi**.

Definizione 2.1.1: Linguaggio

Un linguaggio $\mathscr{L}$ è un sottoinsieme dell'insieme delle stringhe: $\mathscr{L} \subseteq \Sigma^*$.

▷ Quanti sono i linguaggi?
Se la cardinalità di Σ^* è pari alla cardinalità dei numeri naturali, allora la cardinalità dei sottoinsiemi dei numeri naturali è uguale alla cardinalità dei sottoinsiemi degli insiemi delle stringhe:

$$\begin{gathered} |\Sigma^*| = |\mathbb{N}| = \omega \\ \downarrow \\ |2^{\mathbb{N}}| = |2^{\Sigma^*}| = |\mathbb{N} \to \mathbb{N}| = |\mathbb{R}| \end{gathered}$$

Quindi ci sono tanti linguaggi quanti i numeri reali su un alfabeto finito:

$$|\{\mathscr{L} \mid \mathscr{L} \subseteq \Sigma^*\}| \subseteq |\mathbb{R}|$$

Enumerazione delle stringhe

È noto che l'alfabeto sia di cardinalità finita, infatti $|\Sigma| = n$. Si considera una stringa $\sigma \in \Sigma^*$ e occorre associare un numero in modo biunivoco tra i numeri naturali e l'insieme delle stringhe: $\mathbb{N} \leftrightarrow \Sigma^*$.

- Un possibile metodo: una funzione prende un simbolo x e gli passa un valore tra 1 e n:
$$c(x) \in [1, n], \quad x \in \sigma$$
Avendo una stringa $\sigma = x_m x_{m-1}, \ldots, x_1 x_0 \in \Sigma^*$, con lunghezza $m + 1$, qual è il numero biunivoco associato a tale stringa?
Il numero può essere costruito da una funzione g_n che prende una stringa e gli associa un numero naturale $g_n : \Sigma^* \to \mathbb{N}$. Nel dettaglio la funzione è definita come:
$$\begin{aligned} g_n(\sigma) &= g_n(x_m \ldots x_0) \\ &= \sum_{i=0}^{m} c(x_i) \cdot n^i \in \mathbb{N} \end{aligned}$$
Quindi è la sommatoria da $i = 0$ fino a m, cioè per tutti i simboli della stringa, del numero corrispondente a quel simbolo $c(x_i)$ per n^i.

Da questo processo è possibile derivare una stringa dato un numero naturale:

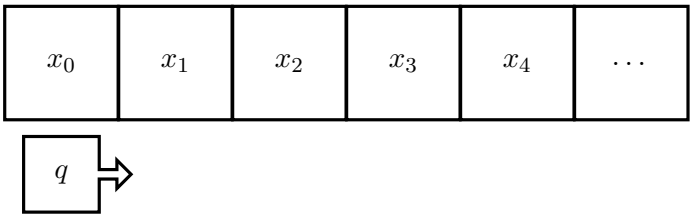
- si divide per n;
- si estrae il rimanente;
- il valore rimanente $c(x_i)$ è associato in modo biunivoco a un simbolo dell'alfabeto.

Esempio

Dato l'alfabeto $\Sigma = \{a, b, c\}$, ad a viene associato il valore 1, a b il valore 2 e a c il valore 3. A questo punto si ricava:
$$g_n(abc) = 1 \cdot 3^2 + 2 \cdot 3^1 + 3 \cdot 3^0 = 18$$

2.2 Automi a stati finiti

L'**automa a stati finiti** è un dispositivo avente un nastro in *input*, suddiviso a blocchi, su cui la macchina non può scrivere, bensì solo leggere:



La stringa σ è formata dai simboli $x_0, x_1, x_2, \ldots$, mentre il **controllore** (dispositivo di lettura) parte da un certo stato q iniziale e può muoversi soltanto da sinistra verso destra. La stringa σ appartiene alle stringhe dell'alfabeto Σ^*.

Quindi si può leggere un simbolo x_i, consumare questo simbolo, effettuare una transizione di stato passando da q a q' e infine passare al simbolo successivo x_{i+1}. Man mano che la testina procede, da sinistra verso destra, essa consuma ciò che è presente sul nastro in *input* fintantoché la stringa σ non si esaurisce. Poiché le stringhe che si possono mettere sul nastro sono finite, allora le operazioni indicate possono continuare fino a un certo numero di passi, dopodiché la macchina termina.
Una descrizione matematica di questa macchina M consiste in:

- un insieme di **stati** Q;
- un insieme di **simboli** Σ;
- una **funzione di transizione** δ;
- uno **stato iniziale** q_0 da cui la macchina parte;
- un insieme di **stati finali** F.

$$M = \langle Q, \Sigma, \delta, q_0, F \rangle$$

Questa descrizione formale corrisponde alla cosiddetta **quintupla**.

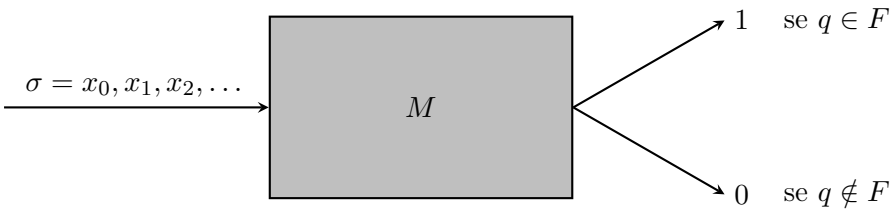
Nel dettaglio questo automa è **deterministico**, indicato con la sigla ASFD, in cui:

- La cardinalità di Q è finita: $|Q| < \omega$.
- La cardinalità dell'alfabeto Σ è finita: $|\Sigma| < \omega$.
- La funzione di transizione δ, in cui dato un *input* specifico restituisce lo stato corrispondente:
  - l'automa si trova in uno stato corrente;
  - legge un simbolo da nastro;
  - genera uno stato. Effettua una transizione di stato: lo stato corrente diventa il nuovo stato calcolato.

$$\delta : Q \times \Sigma \to Q$$

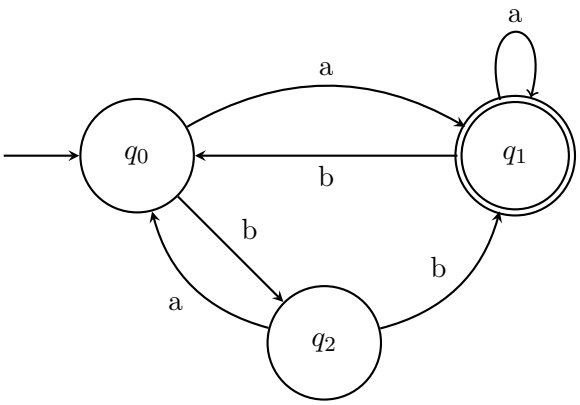
- La macchina inizia da uno stato iniziale q_0 che appartiene agli stati Q.
- Gli stati finali $F \subseteq Q$ sono detti di **accettazione**, in quanto sono stati per i quali la macchina risponde "sì" o "no": se dopo aver finito l'intera sequenza su un nastro ci si ritrova in uno stato dentro F, allora la macchina risponde "sì", altrimenti "no".

Il linguaggio accettato dall'automa è quello per cui dopo aver fatto tutte le transizioni di stato, si conclude in uno stato finale.



Esempio

Un automa è rappresentato graficamente con un grafo:



- La singola freccia entrante che non proviene da alcun nodo indica lo **stato iniziale** dell'automa.
- Il doppio cerchio in un nodo indica che quello è uno **stato finale**.
- Le frecce sono gli archi che a seconda del valore letto (quello sopra l'arco) effettuano una transizione dal nodo cambiando lo stato.

Considerando il precedente grafo dell'automa, si prende in *input* la stringa "abbbaa". L'automa parte dallo stato iniziale q_0 e in successione avvengono i seguenti passi:

- Il primo carattere letto 'a' cambia lo stato da q_0 a q_1.
- Il secondo carattere letto 'b' cambia lo stato da q_1 a q_0.
- Il terzo carattere letto 'b' cambia lo stato da q_0 a q_2.

- Il quarto carattere letto 'b' cambia lo stato da q_2 a q_1.
- Il quinto carattere letto 'a' cambia lo stato da q_1 a sé stesso.
- Infine il sesto carattere letto 'a' cambia lo stato da q_1 a sé stesso.

A questo punto la stringa è conclusa e la macchina restituisce 1 poiché si trova sullo stato q_1, il quale appartiene all'insieme di stati finali F:

$$q_1 \in F \Rightarrow 1$$

Si può considerare δ come la singola transizione e $\hat{\delta}$ come l'intera esecuzione della macchina. $\hat{\delta}$ è definito in modo induttivo:

- **Caso base**: se si è in un certo stato q e non si legge nulla, allora si rimane in q: $\hat{\delta}(q, \varepsilon) = q$.
- **Passo induttivo**: se si è in un certo stato q e si legge σ seguito da un simbolo a, allora corrisponde all'applicazione di δ a partire dallo stato ottenuto dopo n passi aggiungendo il simbolo a:

$$\hat{\delta}(q, \sigma a) = \delta(\hat{\delta}(q, \sigma), a)$$

Dunque la funzione $\hat{\delta}$, prendendo uno stato Q e una stringa, restituisce uno stato come *output*.

$$\begin{cases} \hat{\delta}(q, \varepsilon) = q \\ \hat{\delta}(q, \sigma a) = \delta(\hat{\delta}(q, \sigma), a) \end{cases}$$

Definizione 2.2.2

$\sigma \in \Sigma^*$ è accettata da un ASFD $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ se e solo se $\hat{\delta}$ a partire dallo stato q_0, leggendo tutta σ, appartiene agli stati finali:

$$\hat{\delta}(q_0, \sigma) \in F$$

Il linguaggio accettato dall'automa è l'insieme di tutte le stringhe sull'alfabeto tale che $\hat{\delta}(q_0, \sigma) \in F$:

$$\mathscr{L}(M) = \{\sigma \in \Sigma^* \mid \hat{\delta}(q_0, \sigma) \in F\}$$

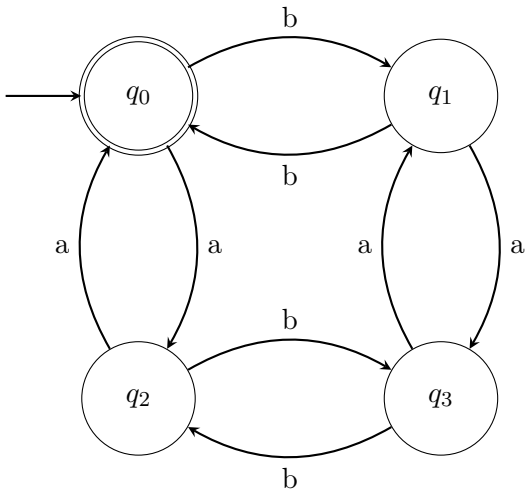
Definizione 2.2.3: Linguaggio regolare

Se M è un automa a stati finiti deterministico, allora $\mathscr{L}(M)$ è detto **regolare**.

2.2.1 Automi deterministici

Esempio

Viene dato il seguente automa a stati finiti:



In questo caso l'alfabeto consiste in:

$$\Sigma = \{\mathrm{a}, \mathrm{b}\}$$

- L'automa possiede quattro stati, perciò è a stati finiti.
- È deterministico perché per ogni stato si ha un solo possibile stato di transizione leggendo un simbolo dal nastro.

▷ La stringa abaaab $\in \mathscr{L}(M)$?
Per verificarlo si percorre l'automa leggendo carattere per carattere la stringa da analizzare:

- Primo carattere 'a': si passa dallo stato iniziale q_0 allo stato q_2.
- Secondo carattere 'b': si passa dallo stato attuale q_2 allo stato q_3.
- Terzo carattere 'a': si passa dallo stato attuale q_3 allo stato q_1.
- Quarto carattere 'a': si passa dallo stato attuale q_1 allo stato q_3.
- Quinto carattere 'a': si passa dallo stato attuale q_3 allo stato q_1.
- Sesto carattere 'b': si passa dallo stato attuale q_1 allo stato q_0.

Quindi la stringa abaaab $\in \mathscr{L}(M)$.

▷ La stringa ababb $\in \mathscr{L}(M)$?
Come in precedenza si percorre l'automa controllando se si conclude la stringa nello stato finale q_0:

- Primo carattere 'a': si passa dallo stato iniziale q_0 allo stato q_2.
- Secondo carattere 'b': si passa dallo stato attuale q_2 allo stato q_3.
- Terzo carattere 'a': si passa dallo stato attuale q_3 allo stato q_1.
- Quarto carattere 'b': si passa dallo stato attuale q_1 allo stato q_0.
- Quinto carattere 'b': si passa dallo stato attuale q_0 allo stato q_1.

Quindi la stringa ababb $\notin \mathscr{L}(M)$, poiché l'automa non conclude la stringa nello stato finale q_0.

Osservazione

Nell'automa appena presentato in esempio sono riconosciuti i seguenti linguaggi:

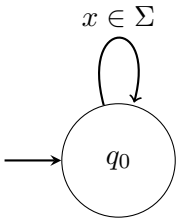
- $\mathscr{L}(M) = \{\varepsilon\}$, perché se la stringa è vuota allora l'automa rimane fermo nello stato finale q_0.
- $\mathscr{L}(M) = \{\sigma \in \Sigma^* \mid \sigma$ contiene un numero pari di a e $b\}$, cioè in σ deve esserci un numero totale sia di 'a' sia di 'b' per tornare in q_0.

$$\mathscr{L}(M) = \{\varepsilon\} \cup \{\sigma \in \Sigma^* \mid \sigma \text{ contiene un numero pari di } a \text{ e } b\}$$

2.2.2 Linguaggi regolari particolari

Esistono altri due linguaggi regolari detti **linguaggi limite**. Questi rappresentano i casi estremi della famiglia dei linguaggi regolari. Il più piccolo linguaggio di tutti e il più grande linguaggio di tutti i linguaggi sono rispettivamente l'**insieme vuoto $\emptyset$** e **tutte le stringhe Σ^***.

▷ Qual è l'automa M tale per cui il linguaggio riconosciuto sia vuoto ($\mathscr{L}(M) = \emptyset$)?
Esistono infiniti automi a stati finiti che hanno come linguaggio riconosciuto l'insieme vuoto. Idealmente l'automa più semplice è quello con un solo stato q_0, non finale, che va su sé stesso per ogni $x \in \Sigma$:



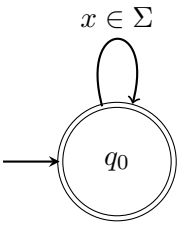
$$M = \langle \{q_0\}, \Sigma, \delta, q_0, \emptyset \rangle$$

Dimostrazione

$$\emptyset = \{\sigma \in \Sigma^* \mid \underbrace{\hat{\delta}_M(q_0, \sigma) \in \emptyset}_{\text{falso}}\}$$

▷ Quale stringa $\sigma \in \Sigma^*$ sta in $\emptyset$?
Nessuna stringa può stare in $\emptyset$, pertanto $\emptyset = \emptyset$.

▷ Qual è l'automa M tale che il linguaggio riconosciuto da M siano tutte le possibili stringhe ($\mathscr{L}(M) = \Sigma^*$)?
In questo caso si hanno infinite stringhe di lunghezza maggiore o uguale a 1 e l'automa che riconosce tutte le stringhe è simile a quello mostrato in precedenza. L'unica differenza è che lo stato q_0 è lo stato finale:



$$M = \langle \{q_0\}, \Sigma, \delta, q_0, \{q_0\} \rangle$$

Dimostrazione

$$\Sigma^* = \{\sigma \in \Sigma^* \mid \underbrace{\hat{\delta}_M(q_0, \sigma) \in \{q_0\}}_{\text{vero}}\}$$

▷ Quale stringa $\sigma \in \Sigma^*$ sta in $\{q_0\}$?
Tutte le stringhe stanno in $\{q_0\}$, pertanto $\Sigma^* = \Sigma^*$.

Osservazione

L'insieme vuoto è sempre contenuto nell'insieme di tutte le stringhe:

$$\emptyset \subseteq \Sigma^*$$

Più in generale l'insieme vuoto è contenuto in qualunque linguaggio, il quale è contenuto per definizione nell'insieme di tutte le stringhe:

$$\emptyset \subseteq \mathscr{L} \subseteq \Sigma^*$$

Tuttavia il fatto che $\mathscr{L}$ sia compreso tra due linguaggi regolari, $\emptyset$ e Σ^*, non implica che il linguaggio $\mathscr{L}$ sia anch'esso regolare. Per verificarlo basta considerare la cardinalità.

L'insieme di tutti i linguaggi $\mathscr{L}$, tale che $\mathscr{L}$ è contenuto nell'insieme di tutte le stringhe, ha la stessa cardinalità dell'insieme di tutti i numeri S, tale che S sia un sottoinsieme dei numeri naturali:

$$|\{\mathscr{L} \mid \mathscr{L} \subseteq \Sigma^*\}| = |\{S \mid S \subseteq \mathbb{N}\}|$$

Perché la cardinalità dell'insieme di tutte le stringhe è uguale alla cardinalità dei numeri naturali:

$$|\Sigma^*| = |\mathbb{N}|$$

Il linguaggio $\mathscr{L}$ è un sottoinsieme di Σ^*, perciò esiste una biiezione che permette di mappare qualunque stringa con un numero in modo biunivoco. Di conseguenza a ogni linguaggio corrisponde un insieme di numeri.
In secondo luogo la cardinalità degli insiemi S è uguale alla cardinalità dei sottoinsiemi di $\mathbb{N}$, che a sua volta con l'ipotesi del continuo è pari alla cardinalità di $\mathbb{R}$:

$$|\{S \mid S \subseteq \mathbb{N}\}| = |2^{\mathbb{N}}| = |\mathbb{R}|$$

Ne consegue che la cardinalità degli insiemi dei linguaggi $\mathscr{L}$ in un alfabeto Σ^* sia uguale alla cardinalità delle funzioni da $\mathbb{N}$ in $\mathbb{N}$:

$$|\{\mathscr{L} \mid \mathscr{L} \subseteq \Sigma^*\}| = |\mathbb{N} \to \mathbb{N}|$$

$\triangleright$ Qual è la cardinalità dei linguaggi regolari?
I **linguaggi regolari** corrispondono all'insieme dei linguaggi $\mathscr{L}$, tale per cui $\mathscr{L}$ sia un sottoinsieme dell'insieme delle stringhe ed esiste un automa a stati finiti deterministico M per cui $\mathscr{L} = \mathscr{L}(M)$:

$$\text{REG} = \{\mathscr{L} \mid \mathscr{L} \subseteq \Sigma^*, \exists M(\text{ASFD}) : \mathscr{L} = \mathscr{L}(M)\}$$

- La prima condizione significa che essere un linguaggio comporta un'infinità non numerabile di linguaggi: sono tanti quanti i numeri reali.
- La seconda condizione sull'esistenza di un ASFD che riconosce quel linguaggio, vincola la cardinalità dei linguaggi regolari.

Pertanto la cardinalità risulta essere:

$$|\text{REG}| = |\mathbb{N}|$$

Da ciò segue che, per il teorema di Cantor, la cardinalità dei linguaggi è strettamente più grande della cardinalità dei linguaggi regolari:

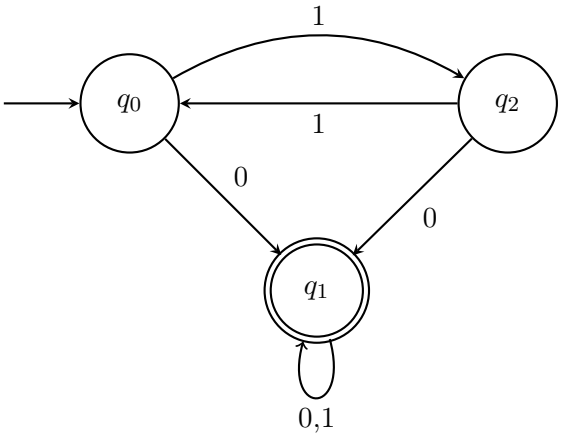
$$|\{\mathscr{L} \mid \mathscr{L} \subseteq \Sigma^*\}| > |\text{REG}|$$

Quindi $\mathscr{L}$ può essere, come anche no, un linguaggio regolare.

Tornando alla relazione $\emptyset \subseteq \mathscr{L} \subseteq \Sigma^*$, dentro a $\mathscr{L}$ ci possono essere tanti linguaggi, ma solo pochi[1] possono essere regolari.

Esempio

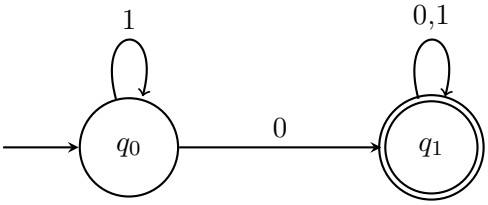
Si considera l'automa $M = \langle \{q_0, q_1, q_2\}, q_0, \{0,1\}, \delta, \{q_1\} \rangle$:



Osservando la topologia di questo automa emerge che per arrivare allo stato finale q_1 bisogna aver letto almeno uno 0; finché si legge 1 si va da q_0 a q_2 e viceversa. Appena si incontra uno 0 si va permanentemente in q_1, perciò il linguaggio è l'insieme delle stringhe appartenenti all'insieme $\{0,1\}$ in cui σ contiene almeno uno 0:

$$\mathscr{L}(M) = \{\sigma \in \{0,1\}^* \mid \sigma \text{ contiene almeno uno } 0\}$$

Tuttavia esiste anche un altro automa che riconosce lo stesso linguaggio. Poiché gli stati q_0 e q_2 sono equivalenti, si può avere il seguente automa M':



Il linguaggio riconosciuto da M è lo stesso di M':

$$\mathscr{L}(M) = \mathscr{L}(M')$$

[1] Sempre una infinità, ma numerabile.

Osservazione

In realtà è possibile sia complicare un automa M, aggiungendo stati equivalenti, sia minimizzarlo al minor numero di stati possibile mediante l'algoritmo di ottimizzazione; M' nell'esempio mostrato è minimo.

Dimostrazione: Uguaglianza di $\mathscr{L}(M)$

Bisogna dimostrare le due inclusioni $\subseteq$ e $\supseteq$ dell'uguaglianza:

$$\mathscr{L} = \underbrace{\{\sigma \in \{0,1\}^* \mid \sigma \text{ contiene almeno uno } 0\}}_{\mathscr{L}}$$

- **Prima inclusione $\mathscr{L} \subseteq \mathscr{L}(M)$**: si prende $x \in \mathscr{L}$, dunque x deve contenere almeno uno 0 e se lo contiene allora $x = v0w$, dove $v, w \in \Sigma^*$ e inoltre v è una sequenza di soli 1 ($v = 1 \dots 1$). È necessario dimostrare che:

  $$\hat{\delta}(q_0, x) \in F = \{q_1\} \iff \hat{\delta}(q_0, x) = 0$$

  Se si parte da q_0 e si legge una sequenza di $1 \dots 1$ e poi uno 0, allora la funzione $\hat{\delta}$ è uguale a q_1:

  $$\hat{\delta}(q_0, \underbrace{1 \dots 1}_{n} 0) = q_1$$

  Ciò si dimostra per induzione:

  - **Caso base**: se $n = 0$, allora $\hat{\delta}(q_0, 0) = q_1$.
  - **Passo induttivo**: se $n > 0$, allora si aggiunge una sequenza di 1:

    $$\hat{\delta}(q_0, \underbrace{1\underbrace{1 \dots 1}_{n}}_{n+1} 0) = q_1$$

    Leggendo sequenze di 1 si rimane negli stati q_0 o q_2 e leggendo poi lo 0 si passa a q_1.

  Allora si ottiene il seguente risultato:

  $$\begin{aligned} \hat{\delta}(q_0, x) &= \hat{\delta}(q_0, v0w), \text{ dove } v = 1 \dots 1 \\ &= \hat{\delta}(q_0, 1 \dots 10w) \\ &= \hat{\delta}(q_1, w) = q_1 \in F \end{aligned}$$

Questo implica che per ogni x, appartenente a $\mathscr{L}$, vale che x appartenga a $\mathscr{L}(M)$, se e solo se $\mathscr{L}$ è contenuto in $\mathscr{L}(M)$:

$$\forall x \in \mathscr{L} : x \in \mathscr{L}(M) \iff \mathscr{L} \subseteq \mathscr{L}(M)$$

- **Seconda inclusione $\mathscr{L} \supseteq \mathscr{L}(M)$**: si assume che x non contenga 0, pertanto $x \notin \mathscr{L}(M)$. Si utilizza l'induzione:
  - **Caso base**: se la lunghezza è $|x| = 0$, allora $\hat{\delta}(q_0, \varepsilon) = q_0 \neq q_1$
  - **Passo induttivo**: se la lunghezza è $0 < |x| \leq n$, allora si aggiunge un carattere finale in modo che $|x1| = n + 1$:

$$\begin{aligned} \hat{\delta}(q_0, x1) &= \delta(\underbrace{\hat{\delta}(q_0, x)}_{\neq q_1}, 1) \\ &= \delta(q_i, 1), \text{ con } i = 0, 2 \\ &\neq q_1 \end{aligned}$$

  Questo implica che per ogni stringa di lunghezza qualsiasi, che non contenga 0, non appartiene al linguaggio riconosciuto dall'automa. Quindi $x \in \mathscr{L}(M)$ e x contiene 0, ciò implica $\mathscr{L} \supseteq \mathscr{L}(M)$.

2.2.3 Automi non deterministici

L'automa di tipo ASFND è un automa a stati finiti **non deterministico**, per cui:

$$M = \langle Q, \Sigma, \delta, q_0, F \rangle$$

L'unico campo a cambiare è la funzione di stato successore δ:

- Negli ASFD è una **funzione.**
- Negli ASFND è una **relazione**.

Si ricorda che le funzioni sono particolari relazioni in cui a ogni *input* la funzione associa un solo risultato. Mentre nelle relazioni, in generale, a ogni *input* ci possono essere più risultati.

Quindi si ha:

$$\delta \subseteq (Q \times \Sigma) \times Q \tag{2.1}$$

Data una coppia:

- la macchina è in uno stato q;

- legge dal nastro il simbolo x;
- a questo punto lo stato q non cambia in un altro stato p andando poi a leggere il simbolo successivo, bensì accade che la macchina si ramifica in nuovi stati $p_1, p_2, \ldots, p_n$.

Avvengono delle transizioni in più stati, come se la macchina si dividesse in più macchine identiche che partono da stati diversi nella lettura del simbolo successivo.

Se l'espressione (2.1) è una **relazione**, allora si può rappresentare δ come:

$$\delta : Q \times \Sigma \to 2^Q$$

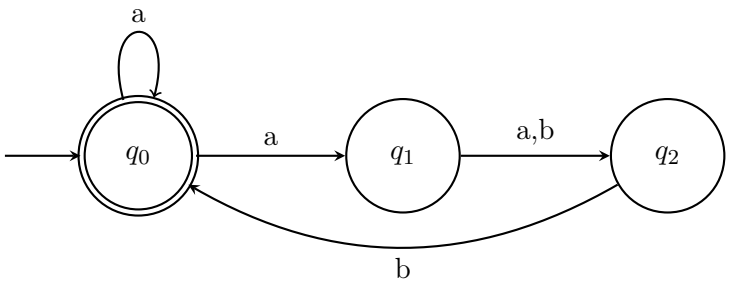
In pratica δ adesso è una **funzione** che va dagli stati Q per i simboli Σ (legge un simbolo dal nastro) e genera un insieme di stati 2^Q.

Se si va nelle parti, cioè in 2^Q, allora $\emptyset \subseteq Q$ e $\delta(q, a) = \emptyset$. Questo significa che nell'automa non deterministico si può non fare passi e rimanere nello stesso stato, viceversa negli automi deterministici è necessario raggiungere un nuovo stato. Inoltre, per ogni stato in Q e ogni simbolo letto in Σ, un'altra proprietà è quella dove la cardinalità degli stati che si può raggiungere è sempre finita:

$$\forall q \in Q, \quad \forall a \in \Sigma : \ |\delta(q, a)| < \omega$$

Esempio

Si considera il seguente automa:



▷ Perché non è deterministico?
Perché in corrispondenza del simbolo 'a' si possono avere più transizioni diverse per lo stesso stato. Perciò questo automa dopo aver letto il simbolo 'a' continua in due strade diverse. Mentre se si è nello stato q_2 si legge 'a', allora la macchina rimane bloccata in q_2 abortendo l'esecuzione.

▷ La stringa abb $\in \mathscr{L}(M)$?
Si percorre l'automa tenendo presente delle possibili ramificazioni, ciascuna indipendente dall'altra:

- Primo carattere 'a': si può rimanere in q_0 oppure effettuare una transizione in q_1.
  - Stato q_0: leggendo il secondo carattere 'b' la macchina abortisce l'esecuzione (si blocca).
  - Stato q_1: leggendo il secondo carattere 'b' si va nello stato q_2. Infine legge l'ultimo carattere 'b' e torna in q_0

Quindi la stringa abb $\in \mathscr{L}(M)$, poiché conclude nello stato finale q_0.

Se esistono più transizioni possibili per un simbolo in lettura, allora l'automa non è deterministico.

La relazione di transizione applicata a più passi consegue nella seguente definizione di $\hat{\delta}$:

$$\begin{cases} \hat{\delta}(q, \varepsilon) = \{q\} \\ \hat{\delta}(q, \sigma a) = \bigcup_{p \in \hat{\delta}(q,\sigma)} \delta(p, a) \end{cases}$$

- Se non si legge nulla, allora si rimane in q generico.
- Se si legge un simbolo, allora si genera un insieme di stati: la macchina si duplica in diversi stati.

Se M è ASFND, cioè

$$M = \langle Q, \Sigma, \delta, q_0, F \rangle,$$

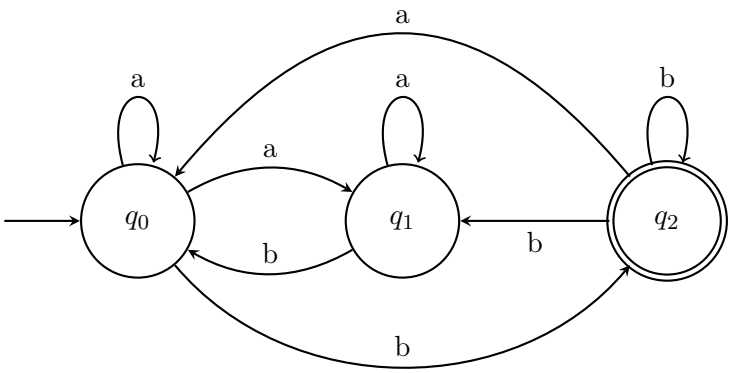
allora il linguaggio riconosciuto è dato da: tutte le stringhe sull'alfabeto tali che l'intersecazione tra gli stati su cui si effettua la transizione e gli insiemi degli stati finali è diverso dall'insieme vuoto:

$$\mathscr{L}(M) = \{\sigma \in \Sigma^* \mid \hat{\delta}(q_0, \sigma) \cap F \neq \emptyset\}$$

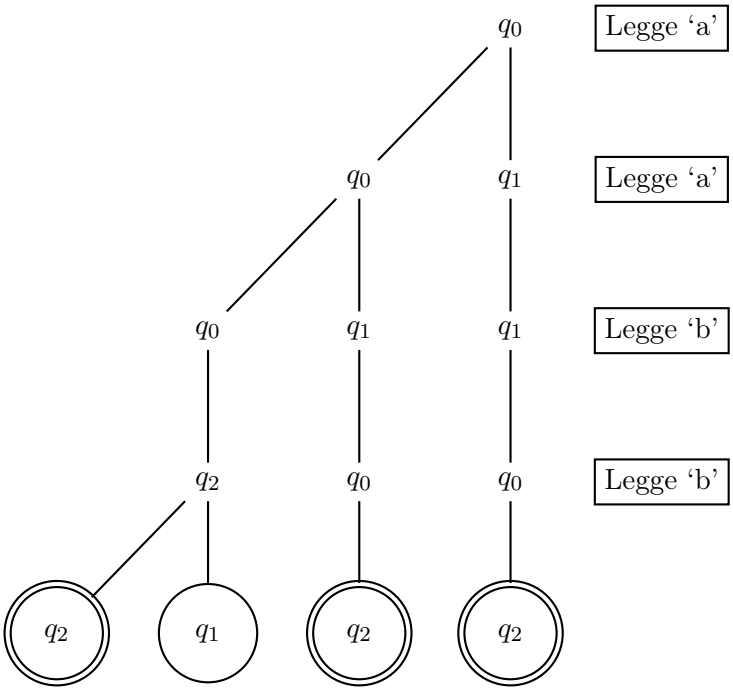
Alla fine dell'intera lettura della stringa si ha un insieme di stati. Se almeno uno di questi stati è finale, allora la stringa è accettata.

Esempio

Si considera l'alfabeto $\Sigma = \{\mathrm{a}, \mathrm{b}\}$ e il seguente automa a stati finiti non deterministico:



$\triangleright$ La stringa aabb $\in \mathscr{L}(M)$?
Si percorre l'automa e tutte le sue ramificazioni:



$$\hat{\delta}(q_0, aabb) = \{q_2, q_1\} \cap \{q_2\} \neq \emptyset$$

Quindi la stringa aabb $\in \mathscr{L}(M)$, poiché almeno uno degli stati raggiunti alla conclusione della stringa è finale.

Osservazione

La computazione su questa tipologia d'automa si sviluppa espandendo un albero di ricerca che prova più stati per ogni simbolo in lettura.

Equivalenza tra ASFD e ASFND

Sebbene gli ASFND abbiano molta più potenza di calcolo, rispetto gli ASFD, la classe dei linguaggi riconosciuti non è più grande della famiglia dei linguaggi regolari.

Il seguente teorema e la sua dimostrazione mostrano che tutto ciò che si può fare con un automa a stati finiti deterministico lo si può anche fare con uno non deterministico. In pratica entrambe le macchine riconoscono la medesima famiglia di linguaggi.

Teorema 2.2.1: Rabin-Scott (1959)

Sia $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ un ASFND. Allora esiste un ASFD chiamata M' tale che $\mathscr{L}(M) = \mathscr{L}(M')$.

Dimostrazione

Occorre verificare entrambe le inclusioni:

- **Prima inclusione** $\subseteq$: un automa a stati finiti deterministico è un caso particolare di un automa a stati finiti non deterministico, dove la funzione di transizione è una relazione di transizione che dato un *input* corrisponde un *output*. Questa osservazione comporta il seguente risultato:

$$\underbrace{\{\mathscr{L} \mid \exists M'(\text{ASFD}) : \mathscr{L} = \mathscr{L}(M')\}}_{\text{REG}} \subseteq \{\mathscr{L} \mid \exists M(\text{ASFND}) : \mathscr{L} = \mathscr{L}(M)\}$$

  Ossia i linguaggi regolari delle ASFD sono contenuti nei linguaggi delle ASFND.

- **Seconda inclusione** $\supseteq$: si costruisce una ASFD con $M' = \langle Q', \Sigma', \delta', q_0', F' \rangle$:
  - La relazione che c'è tra gli stati dell'automa non deterministico e quelli dell'automa deterministico è esponenziale, poiché l'automa non deterministico a ogni computazione legge un simbolo da nastro ed effettua una transizione verso un insieme di stati. Allora l'intuizione sta nel considerare quell'insieme di stati in un nuovo singolo stato, cioè quest'ultimo è rappresentato dall'insieme di stati dell'ASFND. Mentre dall'altra parte gli stati dell'ASFD sono tanti quanti i sottoinsiemi degli stati dell'ASFND. Questo ragionamento comporta che:

$$Q' = 2^Q = \{q_0', \ldots, q_{2^{|Q|}}'\}$$

    Dunque se si hanno 3 stati nell'ASFND, allora se ne hanno 8 nell'ASFD.

- L'alfabeto è uguale per entrambe le macchine:

$$\Sigma' = \Sigma$$

- Lo stato iniziale corrisponde a:

$$q'_0 = \{q_0\}$$

- Gli stati finali sono un qualunque insieme S di stati in Q tale che S contenga almeno uno stato finale dell'automa non deterministico:

$$F' = \{S \subseteq Q \mid S \cap F \neq \emptyset\}$$

- La funzione di transizione δ' dell'ASFD dell'insieme di stati S che legge un simbolo a è uguale all'unione di tutti gli stati q appartenenti a S della funzione δ in q che legge a:

$$\delta'(S, a) = \bigcup_{q \in S} \delta(q, a)$$

Occorre dimostrare che, per ogni stringa σ appartenente all'insieme di tutte le stringhe Σ^*, con entrambe le macchine si arriva allo stesso insieme di stati:

$$\forall \sigma \in \Sigma^* : \hat{\delta}(q_0, \sigma) = \hat{\delta}'(q'_0, \sigma)$$

Si effettua l'induzione sulla lunghezza di σ:

- **Caso base**: $|\sigma| = 0$, cioè $\sigma = \varepsilon$. Allora succede che:

$$\hat{\delta}(q_0, \varepsilon) = \{q_0\} = q'_0 = \hat{\delta}'(q'_0, \varepsilon)$$

- **Passo induttivo**: $0 < |\sigma| \leq n$, cioè $\sigma \neq \varepsilon$. Allora per ogni σ appartenente all'insieme di tutte le stringhe Σ^*, tali per cui la cardinalità di σ è minore o uguale di n, si ha lo stesso insieme di stati:

$$\forall \sigma \in \Sigma^*.|\sigma| \leq n : \hat{\delta}(q_0, \sigma) = \hat{\delta}'(q'_0, \sigma) \tag{2.2}$$

  Si prende una generica stringa σ lunga n e un simbolo a appartenente all'alfabeto Σ, ne consegue che la lunghezza di σa è pari a $n+1$. Quindi

si mostra che l'asserzione (2.2) sia vera per σa:

$$\begin{aligned}
\hat{\delta}'(q_0', \sigma a) &= \hat{\delta}'(\{q_0\}, \sigma a) \\
&= \delta'(\hat{\delta}'(\{q_0\}, \sigma), a) \\
\left(\begin{smallmatrix}\text{Ipotesi}\\ \text{induttiva}\end{smallmatrix}\right) &= \delta'(\underbrace{\hat{\delta}(q_0, \sigma)}_{S}, a) \\
&= \bigcup_{p \in \hat{\delta}(q_0, \sigma)} \delta(p, a) \\
&= \hat{\delta}(q_0, \sigma a)
\end{aligned}$$

Di conseguenza è dimostrata l'asserzione (2.2).

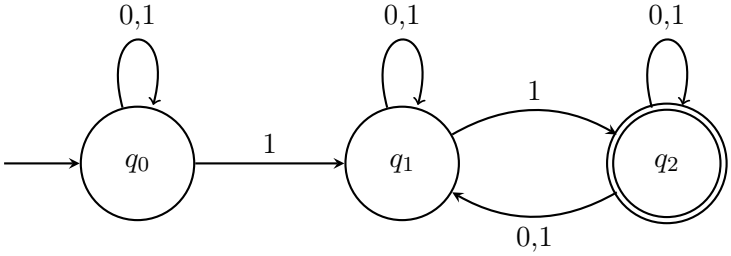
In conclusione si prende la stringa $\sigma \in \overbrace{\mathscr{L}(M)}^{\text{ASFND}}$, che è riconosciuta dall'automa se e solo se $\hat{\delta}(q_0, \sigma) \cap F \neq \emptyset$ e poiché è verificata l'espressione (2.2), allora l'automa riconosce la stringa soltanto se $\hat{\delta}'(q_0', \sigma) \cap F \neq \emptyset$.
Sapendo che $F' = \{S \subseteq Q \mid S \cap F \neq \emptyset\}$, allora σ è riconosciuta soltanto se $\hat{\delta}'(q_0, \sigma) \in F'$ e questo è vero quando nell'ASFD $\sigma \in \mathscr{L}(M)$.

$$\begin{aligned}
\sigma \in \overbrace{\mathscr{L}(M)}^{\text{ASFND}} &\iff \hat{\delta}(q_0, \sigma) \cap F \neq \emptyset \\
&\iff \hat{\delta}'(q_0', \sigma) \cap F \neq \emptyset \\
&\iff \hat{\delta}'(q_0, \sigma) \in F' \\
&\iff \sigma \in \underbrace{\mathscr{L}(M')}_{\text{ASFD}}
\end{aligned}$$

Per ogni ASFND esiste un ASFD equivalente, il quale possiede un numero di stati esponenziale rispetto l'ASFND.

Esempio

Si considera un alfabeto $\Sigma = \{0, 1\}$ e una macchina M di tipo ASFND:



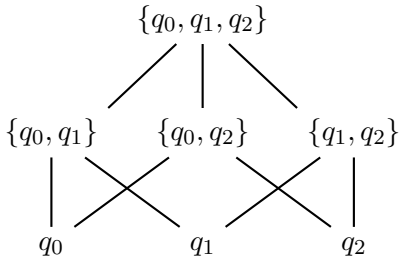
Per ricavare l'automa deterministico viene applicata la **tecnica di Rabin-Scott**.

Si costruisce un automa avente:

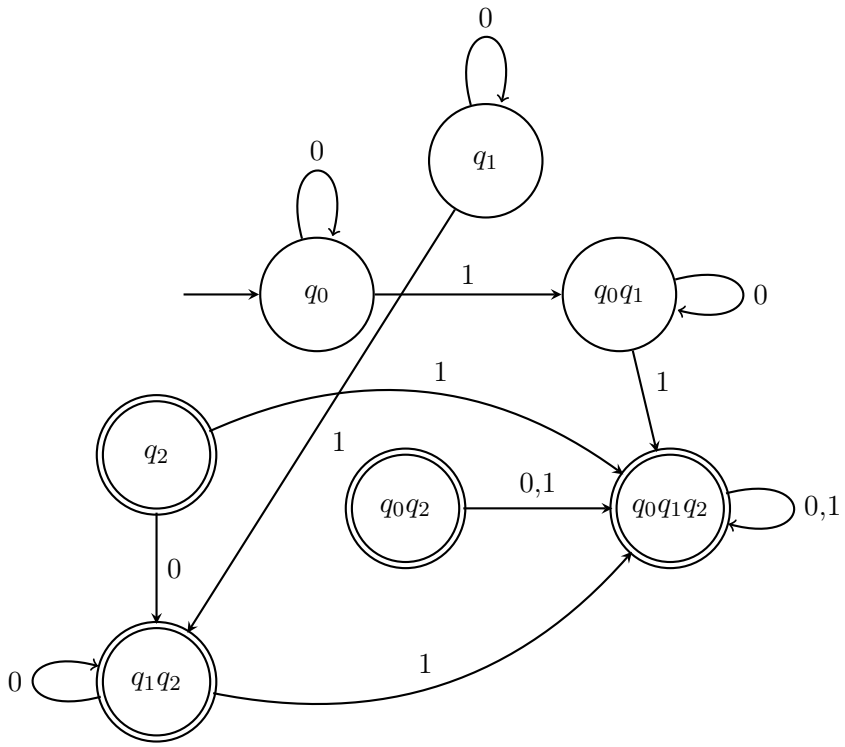
- Gli stati stati sono le parti di $\{q_0, q_1, q_2\}$.
- L'alfabeto è sempre $\{0, 1\}$.
- La funzione δ' verrà specificata successivamente.
- Lo stato F' è come quello definito in precedenza.

$$M' = \langle 2^{\{q_0,q_1,q_2\}}, \{0,1\}, \delta', \{q_0\}, F' \rangle$$

L'insieme $Q' = 2^{\{q_0,q_1,q_2\}}$ corrisponde a:

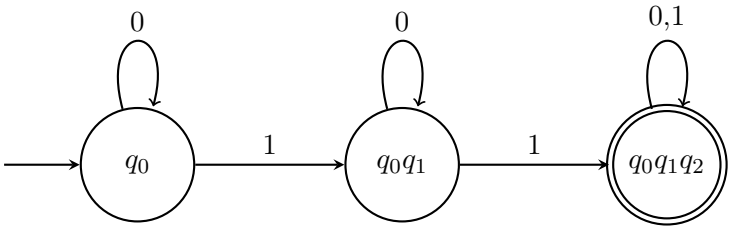


L'automa deterministico è costruito seguendo la parallelizzazione nell'ASFND:



Per ogni stato c'è una sola transizione, di conseguenza è ASFD. Tuttavia è **ridondante**, ovvero alcuni stati non sono mai raggiungibili: $\{q_1\}$, $\{q_2\}$, $\{q_1q_2\}$, $\{q_0q_2\}$.

Essendo che la macchina non potrà mai trovarsi in questi stati, allora sono rimovibili. L'automa ASFD risultante è il seguente:



Le stringhe accettate sono sequenze di 0, seguite da un 1, seguito da sequenze di 0, seguite da un 1 e infine seguito da una qualunque stringa binaria:

$$0^*10^*1\sigma,\ \sigma \in \{0,1\}^*$$

2.2.4 Automi non deterministici con ε-transizioni

Viene aggiunto agli ASFND un **ε-transizione**. Nel dettaglio si tratta di una transizione che avviene senza leggere alcun simbolo, ovvero l'automa può andare in più stati senza operare la lettura di un simbolo dal dispositivo di ingresso.
Un ASFND con ε-transizioni è definito come:

$$M = \langle Q, \Sigma, \delta, q_0, F \rangle$$

dove δ possiede la seguente caratteristica:

- la macchina è in uno stato Q;
- legge in *input* un simbolo di Σ oppure nulla $\{\varepsilon\}$;
- questo va nelle parti di Q.

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$$

Di conseguenza anche $\hat{\delta}$ è definita diversamente e occorre conoscere il concetto di **ε-chiusura**.

Definizione 2.2.4: ε-chiusura

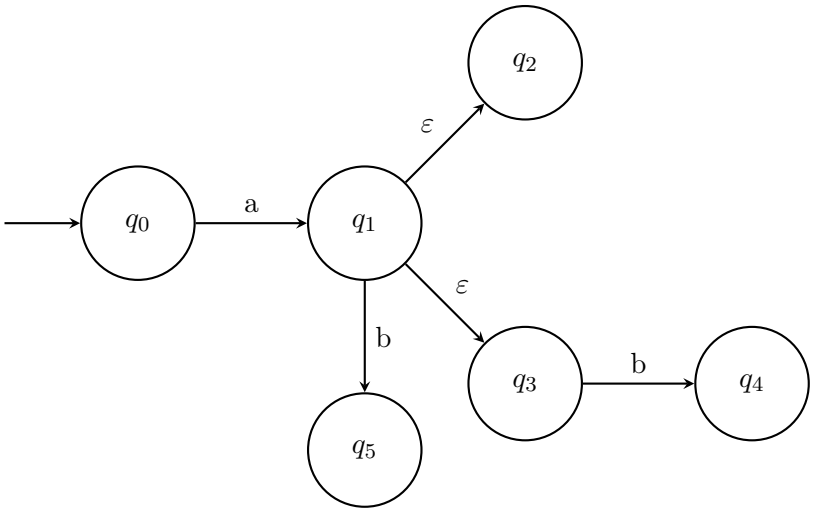
Se uno stato $p \in Q$, allora l'ε-chiusura di p è dato dall'insieme di tutti gli stati $q \in Q$ tali che da p, con un certo numero di passi etichettati ε, permettono di arrivare a q:

$$\varepsilon\text{-chiusura}(p) = \{q \in Q \mid p \overset{\varepsilon}{\rightsquigarrow}{}^* q\}$$

L'ε-chiusura è l'insieme di stati che vengono percorsi leggendo ε.

Esempio

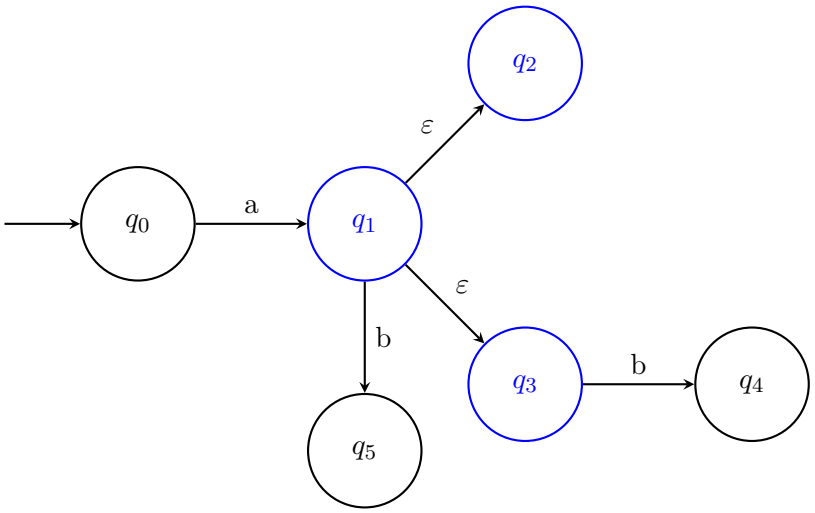
Si suppone di avere la seguente ε-ASFND:



$\triangleright$ Qual è l'ε-chiusura?

- L'ε-chiusura di q_0 è q_0: ε-chiusura$(q_0) = \{q_0\}$. Cioè da q_0 leggendo ε si rimane in q_0.
- L'ε-chiusura di q_1 permette di ottenere $\{q_1, q_2, q_3\}$: ε-chiusura$(q_1) = \{q_1, q_2, q_3\}$. Quindi ci sono degli stati che possono essere raggiunti con ε.

Bisogna considerare tutti gli stati che si raggiungono in questo modo:



Successivamente leggendo il carattere 'b' ne consegue che:

$$q_1 \rightarrow q_5$$
$$q_2 \rightarrow \text{abortisce}$$
$$q_3 \rightarrow q_4$$

Per induzione si definisce $\hat{\delta}$:

$$\begin{cases} \hat{\delta}(q, \varepsilon) = \varepsilon\text{-chiusura}(q) \\ \hat{\delta}(q, \sigma a) = \bigcup_{p \in \hat{\delta}(q,\sigma)} \varepsilon\text{-chiusura}(\delta(p, a)) \end{cases} \tag{2.3}$$

▷ Quale sarebbe l'ε-chiusura leggendo 'a' nell'automa in esempio?

$$\begin{aligned} \hat{\delta}(q_0, \mathrm{a}) &= \bigcup_{p \in \underbrace{\hat{\delta}(q_0, \varepsilon)}_{\{q_0\}}} \varepsilon\text{-chiusura}(\delta(p, \mathrm{a})) \\ &= \varepsilon\text{-chiusura}(\underbrace{\delta(q_0, \mathrm{a})}_{\{q_1\}}) \\ &= \{q_1, q_2, q_3\} \end{aligned}$$

Partendo a leggere il simbolo 'a' l'automa esegue le transizioni $\{q_1, q_2, q_3\}$.

Osservazione

In questo modo è possibile rappresentare delle computazioni interne all'automa. In corrispondenza di un simbolo letto esegue delle computazioni interne (transizioni di stato).

Teorema d'equivalenza tra ε-ASFND e ASFND

Anche aggiungendo le ε-transizioni si rimane nei linguaggi regolari.

Osservazione

Un automa non deterministico senza ε-transizioni è un caso particolare di automa con ε-transizioni.

Teorema 2.2.2: Equivalenza tra ε-ASFND e ASFND

Sia $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ un ASFND con ε-transizioni. Allora esiste un M' ASFND senza ε-transizioni tale che:

$$\mathscr{L}(M) = \mathscr{L}(M')$$

Dimostrazione

Si costruisce il seguente automa senza ε-transizioni:

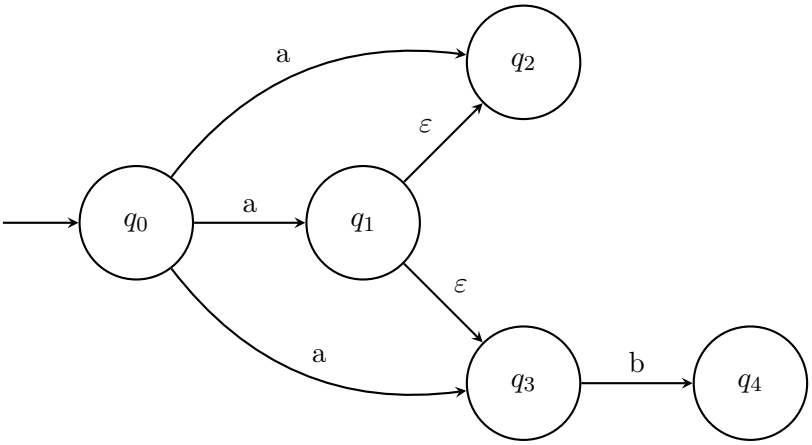
$$M' = \langle Q', \Sigma', \delta', q_0', F' \rangle$$

- Q' è uguale a Q, cioè non serve aggiungere nuovi stati.
- Σ' è lo stesso di Σ.

- Lo stato iniziale q_0' rimane il medesimo q_0: $q_0' = q_0$.
- La relazione di transizione δ' di uno stato q leggendo un simbolo a è uguale alla $\hat{\delta}$ dello stato q leggendo a:

$$\delta'(q, a) = \hat{\delta}(q, a) = \varepsilon\text{-chiusura}(\delta(q, a))$$

In riferimento all'esempio precedente (senza considerare lo stato q_5), al posto di effettuare una transizione verso q_1 e poi leggendo ε delle transizioni verso q_2 e q_3, si esegue una transizione direttamente da q_0 a q_2 e q_3 leggendo 'a'.



In pratica avviene l'ε-chiusura, cioè tutte le transizioni che avvengono in un certo stato leggendo ε sono tutte effettuate leggendo il simbolo 'a'.

- Anche F' cambia nella seguente sua definizione:

$$F' = \begin{cases} F \cup \{q_0\} & \text{se } \varepsilon\text{-chiusura}(q_0) \cap F \neq \emptyset \\ F & \text{altrimenti} \end{cases}$$

Dentro F' c'è lo stato iniziale se e solo se leggendo ε e partendo da q_0, si arriva in uno stato finale. Allora questo stato iniziale, che diventa quello finale, entra negli stati di accettazione. Altrimenti F' è esattamente F.

Occorre dimostrare che per ogni $x \in \Sigma^*$:

$$\underbrace{\hat{\delta}(q_0, x) \cap F \neq \emptyset}_{x \in \mathscr{L}(M)} \iff \underbrace{\hat{\delta}'(q_0', x) \cap F' \neq \emptyset}_{x \in \mathscr{L}(M')} \tag{2.4}$$

- **Prima implicazione $\Rightarrow$**: se ε-chiusura$(q_0) \cap F \neq \emptyset$, allora per definizione di F' vale che $q_0 \in F'$. Quindi $\{q_0\} \cap F' \neq \emptyset$.
- **Seconda implicazione $\Leftarrow$**: si utilizza l'induzione sulle stringhe x che stanno in Σ^*:

– **Caso base**: con $|x| = 1$, allora $x = a$ per qualche simbolo $a \in \Sigma$. Tuttavia per definizione si ha:

$$\begin{aligned}\hat{\delta}'(q_0', a) &= \delta'(q_0', a) \\ &= \hat{\delta}(q_0, a)\end{aligned}$$

– **Passo induttivo**: con $|x| > 1$, se per ogni stringa nell'insieme delle stringhe la lunghezza è compresa in $1 \leq |x| \leq n$, allora vale la seguente asserzione:

$$\hat{\delta}'(q_0', x) = \hat{\delta}(q_0, x) \qquad \text{(ipotesi induttiva)}$$

> Quest'ultima relazione è molto più forte rispetto la doppia implicazione:
>
> $$\hat{\delta}(q_0, x) \cap F \neq \emptyset \iff \hat{\delta}'(q_0', x) \cap F' \neq \emptyset$$
>
> Perché si sta dimostrando per induzione che direttamente:
>
> $$\hat{\delta}(q_0, x) = \hat{\delta}'(q_0', x)$$

Si considera un simbolo $a \in \Sigma$, così da verificare il caso $n + 1$ nella definizione di $\hat{\delta}'$:

$$\hat{\delta}'(q_0', \underbrace{xa}_{n+1}) \doteq \bigcup_{p \in \hat{\delta}'(q_0', x)} \delta'(p, a)$$

δ' è la $\hat{\delta}(q, a)$ vista nella descrizione della quintupla M':

$$\bigcup_{p \in \hat{\delta}'(q_0', x)} \delta'(p, a) \doteq \bigcup_{p \in \hat{\delta}'(q_0', x)} \hat{\delta}(p, a)$$

Tramite l'ipotesi induttiva si cambia l'unione:

$$\bigcup_{p \in \hat{\delta}'(q_0', x)} \hat{\delta}(p, a) \overset{\text{I.I.}}{=} \bigcup_{p \in \hat{\delta}(q_0, x)} \hat{\delta}(p, a)$$

Si sostituisce per la prima definizione di $\hat{\delta}$ in (2.3):

$$\bigcup_{p \in \hat{\delta}(q_0, x)} \hat{\delta}(p, a) \doteq \bigcup_{p \in \hat{\delta}(q_0, x)} \varepsilon\text{-chiusura}(\delta(p, a))$$

Infine per la seconda definizione di $\hat{\delta}$ in (2.3) si ottiene:

$$\bigcup_{p \in \hat{\delta}(q_0, x)} \varepsilon\text{-chiusura}(\delta(p, a)) \doteq \hat{\delta}(q_0, xa)$$

Quindi è dimostrato che $\forall x \in \Sigma^*$:

$$\hat{\delta}(q_0, x) = \hat{\delta}'(q_0', x) \tag{2.5}$$

$\triangleright$ Se risulta vera la relazione (2.5), allora lo è anche l'espressione (2.4)?
In generale F non è uguale a F', tuttavia dato che F' corrisponde a

$$F' = \begin{cases} F \cup \{q_0\} & \varepsilon\text{-chiusura}(q_0) \cap F \neq \emptyset \\ F & \text{altrimenti} \end{cases}$$

allora si nota che F è sempre contenuto in F', ne consegue che in (2.4) è vera l'implicazione $\Rightarrow$. Mentre per l'altra implicazione $\Leftarrow$ si suppone che $F' \neq F$:

$$\begin{aligned} F' \neq F &\Leftrightarrow F' = F \cup \{q_0\} \\ &\Leftrightarrow \varepsilon\text{-chiusura} \cap F \neq \emptyset \end{aligned}$$

Supponendo per assurdo falsa $\hat{\delta}'(q_0', x) \cap F' \neq \emptyset$ nell'espressione (2.4), allora l'unico punto in comune sarebbe q_0 che non appartiene a F:

$$\hat{\delta}'(q_0', x) \cap F' = \{q_0\} \ \wedge \ \varepsilon\text{-chiusura}(q_0) \cap F \neq \emptyset$$

Poiché $\hat{\delta}'(q_0', x)$ genera l'ε-chiusura e q_0 è il risultato dell'intersezione appena descritta, allora l'ε-chiusura(q_0) è contenuta nell'insieme $\hat{\delta}'(q_0', x)$. Se sta in questo insieme, allora è presente anche nell'insieme a esso equivalente, cioè $\hat{\delta}(q_0, x)$.
Quindi si deriva che:

$$\hat{\delta}(q_0, x) \cap F \neq \emptyset, \quad \text{con } \varepsilon\text{-chiusura}(q_0) \subseteq \hat{\delta}'(q_0', x) \equiv \hat{\delta}(q_0, x)$$

Ecco che l'implicazione $\Leftarrow$ è dimostrata.

2.3 Espressioni regolari

Le **espressioni regolari** sono tecniche che consentono di esprimere cosa è contenuto all'interno dei linguaggi regolari:

- **Casi base**:
  - $\emptyset$ è un'espressione regolare.
  - ε è un'espressione regolare.
  - Per ogni simbolo appartenente all'alfabeto $a \in \Sigma$, il simbolo a è un'espressione regolare.
- **Caso induttivo**:
  - Se si hanno r, s espressioni regolari, allora:

    $$r + s, \qquad r \cdot s, \qquad r^*$$

    Sono anche esse espressioni regolari.

Il significato delle espressioni regolari è fornito tramite una funzione di interpretazione `Exp`:

- Il significato dell'insieme vuoto è l'insieme vuoto:

$$\texttt{Exp}(\emptyset) = \emptyset$$

- Il significato di ε è un linguaggio composto solo dalla stringa ε:

$$\texttt{Exp}(\varepsilon) = \{\varepsilon\}$$

- Il significato di un simbolo a è un linguaggio composto soltanto da quel simbolo:

$$\texttt{Exp}(a) = \{a\}$$

`Exp` è un'espressione regolare definita induttivamente, la quale restituisce un linguaggio, ovvero 2^{Σ^*} (un insieme di Σ^*).

A questo punto si compongono i successivi linguaggi in modo induttivo:

- Date le espressioni e_1, e_2, il **linguaggio unione** è definito come:

$$\texttt{Exp}(e_1 + e_2) = \texttt{Exp}(e_1) \cup \texttt{Exp}(e_2)$$

- Date le espressioni e_1, e_2, il **linguaggio concatenazione** è definito come:

$$\texttt{Exp}(e_1 \cdot e_2) = \texttt{Exp}(e_1) \cdot \texttt{Exp}(e_2)$$

- Data l'espressione regolare e^*, il **linguaggio iterazione** è definito come:

$$\texttt{Exp}(e^*) = \texttt{Exp}(e)^*$$

2.3.1 Proprietà delle espressioni regolari

Nelle espressioni regolari sono valide diverse proprietà:

- **Distributiva sinistra**:

$$e_1 \cdot (e_2 + e_3) = e_1e_2 + e_1e_3$$

- **Distributiva destra**:

$$(e_1 + e_2) \cdot e_3 = e_1e_3 + e_2e_3$$

- **Associativa**:

$$e_1 + (e_2 + e_3) = (e_1 + e_2) + e_3$$
$$e_1 \cdot (e_2 \cdot e_3) = (e_1 \cdot e_2) \cdot e_3$$

- **Elemento neutro**:
$$\varepsilon e = e\varepsilon = e$$
- **Elemento vuoto**:
$$\emptyset \cdot e = e \cdot \emptyset = \emptyset$$
- **Unione di due linguaggi medesimi**:
$$e + e = e$$
- **Iterazione del vuoto**:
$$\emptyset^* = \varepsilon$$
- **Assorbimento dell'iterazione di e^***:
$$e^* + e = e^*$$
- **Iterazione dell'iterazione di e^***:
$$(e^*)^* = e^*$$
- **Iterazione dell'unione di linguaggi**:
$$(e_1 + e_2)^* = (e_1^* \cdot e^*)^*$$
- **Unione di un linguaggio e del vuoto**:
$$e + \emptyset = e$$

2.3.2 Corrispondenza tra espressioni regolari e ASFD

Teorema 2.3.3: McNaughton-Yamada (1960)

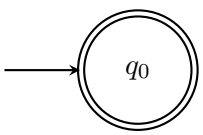
Sia r un'espressione regolare. Allora esiste un M ε-ASFND tale che il linguaggio da esso riconosciuto sia uguale al linguaggio dell'espressione regolare r.

$$\exists M(\varepsilon\text{-ASFND}) : \mathscr{L}(M) = \texttt{Exp}(r)$$

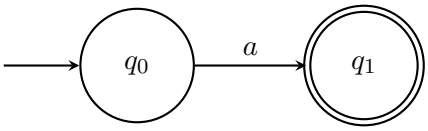
Dimostrazione

Presa un'espressione regolare r che appartiene a `Exp.REG`, bisogna costruire un automa M di tipo ε-ASFND tale che $\mathscr{L}(M) = \texttt{Exp}(r)$.

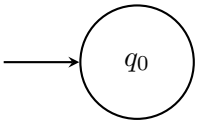
- **Caso base**:
  - ε è riconosciuto dall'ε-ASFND:



– Preso $a \in \Sigma$, allora a è riconosciuto da un ε-ASFND:

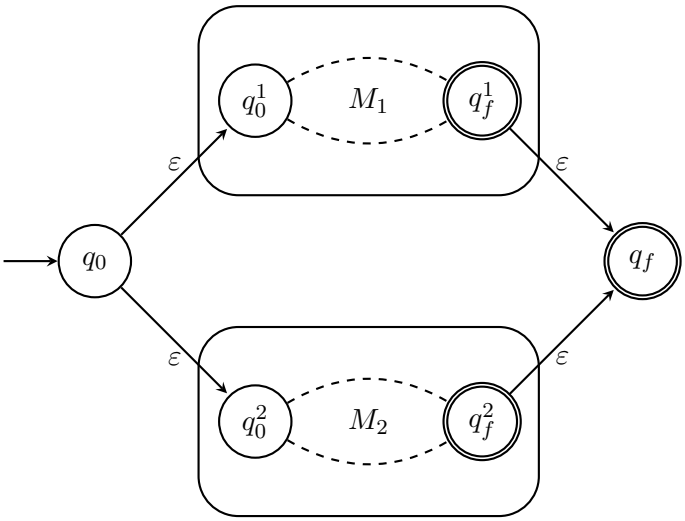


– Preso $\emptyset$, allora $\emptyset$ è riconosciuto da:



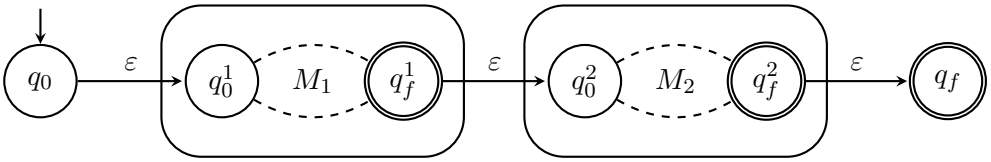
- **Passo induttivo**: l'ipotesi induttiva è che r_1, r_2 siano espressioni regolari con automi equivalenti rispettivamente M_1, M_2.

  – L'automa $r_1 + r_2$ è la composizione parallela dei due automi. Si prova a vedere che se una macchina tra M_1 e M_2 riesce ad accettare la stringa, allora tale stringa sta nell'unione dei due linguaggi:

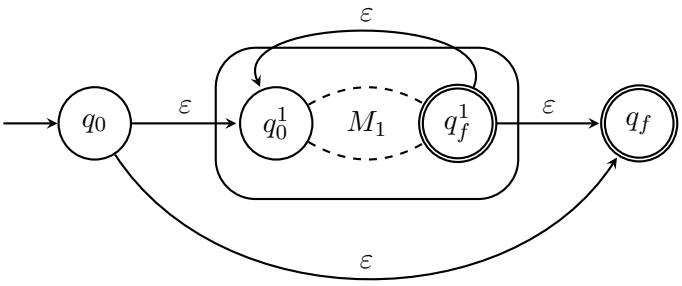


L'automa che riconosce l'unione di due linguaggi è la composizione parallela dei due automi.

  – L'automa $r_1 \cdot r_2$ è la composizione sequenziale della M_1 seguita dalla M_2:



– L'automa r_1^* è definito come:



Da q_0 con ε si attiva la macchina M_1. Se si raggiunge q_f^1 è necessario riconcatenare il linguaggio r_1 all'inizio della macchina e inoltre si effettua una ε-transizione in q_f con il quale si può uscire.
Infine la ε-transizione tra q_0 e q_f permette di considerare la stringa vuota presente in r_1^* (altrimenti si avrebbe r_1^+, cioè $r_1^* \setminus \{\varepsilon\}$).

Questo teorema afferma che ogni volta in cui si ha un'espressione regolare, allora si ha un linguaggio regolare:

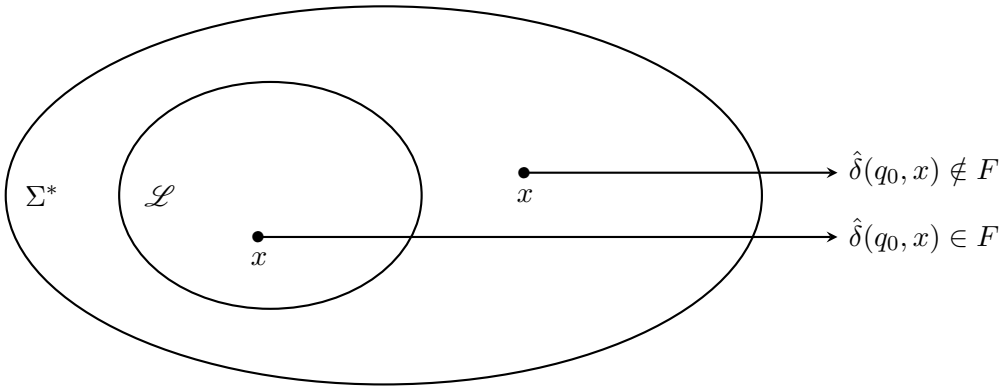
$$\texttt{Exp.REG} \iff \texttt{Ling.REG}$$

2.3.3 Proprietà di chiusura dei linguaggi regolari

La prima conseguenza derivante dalla definizione di **linguaggio regolare** è che: i linguaggi regolari sono in generale **chiusi**, ovvero ci sono delle operazioni che prendono un linguaggio regolare e lo trasformano in un altro linguaggio regolare. Le operazioni in questione sono quelle delle **espressioni regolari**: unione, iterazione e concatenazione.

$$\cup, \cdot : \text{REG} \times \text{REG} \to \text{REG}$$
$$* : \text{REG} \to \text{REG}$$

Altre proprietà derivano da questa osservazione: se $\mathscr{L} \in \text{REG} \subseteq 2^{\Sigma^*}$, allora per il complemento di $\mathscr{L}$, cioè $\overline{\mathscr{L}} = \Sigma^* \setminus \mathscr{L}$, esiste un ASFD tale che riconosca questo $\overline{\mathscr{L}}$?
Quando si costruisce un automa deterministico M', occorre rispettare la condizione per cui $\hat{\delta}(q_0, x) \notin F$:



Ogni stato non finale di un automa, che riconosce $\mathscr{L}$, è uno stato finale per un automa che riconosce $\overline{\mathscr{L}}$:

$$M = \langle Q, \Sigma, \delta, q_0, F \rangle \quad \text{riconosce } \mathscr{L}$$
$$M' = \langle Q, \Sigma, \delta, q_0, F' \rangle \quad \text{riconosce } \overline{\mathscr{L}}$$

dove $F' = Q \setminus F$. Pertanto $\overline{\mathscr{L}}$ si può vedere come:

$$\overline{\mathscr{L}} = \Sigma^* \setminus \mathscr{L} = \mathscr{L}(M')$$

Applicando la **legge di De Morgan** si ottiene che i linguaggi regolari sono **chiusi per intersezione**:

$$\mathscr{L}_1, \mathscr{L}_2 \in \text{REG} \Rightarrow \mathscr{L}_1 \cap \mathscr{L}_2 = \text{REG}$$

Dimostrazione

Per De Morgan l'intersezione tra $\mathscr{L}_1$ e $\mathscr{L}_2$ è uguale al complemento del complemento di $\mathscr{L}_1$ unito al complemento di $\mathscr{L}_2$:

$$\mathscr{L}_1 \cap \mathscr{L}_2 = \overline{(\overline{\mathscr{L}_1} \cup \overline{\mathscr{L}_2})} \in \text{REG}$$

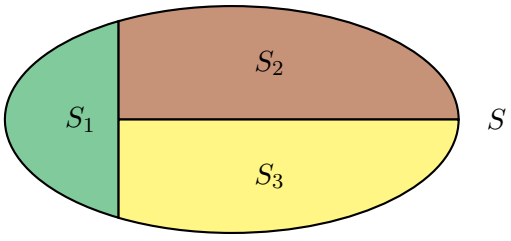
2.4 Automa minimo

Prima di trattare l'**automa minimo** è necessario affrontare vari concetti:

- **Partizione di un insieme** S: corrisponde a tanti sottoinsiemi $S_1, \ldots, S_n$ tali che:
  - Per ogni $i = 1, \ldots, n$: $S_i \subseteq S$.
  - Se si uniscono tutti gli S_i, allora si ottiene S:

$$\bigcup_{i=1}^{n} S_i = S$$

  - Per ogni $i \neq j$: $S_i \cap S_j \neq \emptyset$



- **Relazione di equivalenza**: se la relazione $\sim$ è di equivalenza, ossia è un sottoinsieme di $S \times S$, allora per ogni $a \in S$ si ha che la classe di equivalenza di un oggetto, rispetto la relazione $\sim$, è un insieme $b \in S$ tale che a è equivalente a b:

$$[a]_{\sim} \subseteq S$$

Se si considera la collezione di tutte le classi di equivalenza tale che $a \in S$, allora è una partizione di S: $\{[a]_{\sim} \mid a \in S\}$ è una partizione di S.

- **Indice finito**: si dice che la relazione $\sim$ sia l'indice finito quando ha una cardinalità finita, ovvero se partiziona l'insieme S in un numero finito di classi di equivalenza:

$$|\{[a]_{\sim} : a \in S\}| < \omega$$

Relazione raffinata di equivalenza: è un raffinamento della relazione $\sim$, se per ogni classe di equivalenza $[a]_{\approx}$ questa è sottoinsieme di $[b]_{\sim}$:

$$[a]_{\approx} \subseteq [b]_{\sim}$$

2.4.1 Relazioni tra linguaggi

- Dato un linguaggio $\mathscr{L} \subseteq \Sigma^*$, si definisce la **relazione indotta dal linguaggio** $\mathscr{R}_{\mathscr{L}}$ come sottoinsieme di $\Sigma^* \times \Sigma^*$:

$$\mathscr{R}_{\mathscr{L}} \subseteq \Sigma^* \times \Sigma^*$$

Una parola x è in **relazione** $\mathscr{L}$ con una parola y se e solo se, per ogni parola $z \in \Sigma^*$, si aggiunge a destra di x la z e questa appartiene a $\mathscr{L}$ se e solo se y con l'aggiunta a destra di z appartiene a $\mathscr{L}$:

$$x \mathscr{R}_{\mathscr{L}} y \overset{\Delta}{\iff} \forall z \in \Sigma^* : xz \in \mathscr{L} \Leftrightarrow yz \in \mathscr{L} \tag{2.6}$$

- Si suppone di aver un automa M a stati finiti deterministico:

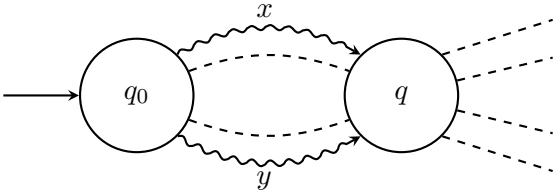
$$M = \langle Q, \Sigma, \delta, q_0, F \rangle$$

Si definisce la **relazione indotta dall'automa** M come sottoinsieme di $\Sigma^* \times \Sigma^*$:

$$\mathscr{R}_M \subseteq \Sigma^* \times \Sigma^*$$

Una parola x è in **relazione** M con y solo se partendo da uno stato q_0, leggendo x, si arriva allo stesso stato q_0 leggendo y:

$$x \mathscr{R}_M y \overset{\Delta}{\iff} \hat{\delta}(q_0, x) = \hat{\delta}(q_0, y)$$



Quindi si può affermare che x e y sono equivalenti.

- Una relazione sui linguaggi $\mathscr{R} \subseteq \Sigma^* \times \Sigma^*$ è **invariante destra** se:

$$x \mathscr{R} y \Rightarrow \forall z \in \Sigma^* : xz \mathscr{R} yz$$

▷ Se tutte le volte in cui si hanno due stringhe in relazione, si ottengono ancora due stringhe sempre in relazione completandole entrambe a destra con una medesima stringa, allora le relazioni indotte dal linguaggio e dall'automa sono anch'esse invarianti destre?

Lemma 2.4.1

Sia $\mathscr{R}_{\mathscr{L}}$ e sia $\mathscr{R}_M$ sono invarianti destra.

Dimostrazione

▶ $\mathscr{R}_{\mathscr{L}}$ è invariante destra:

- Si considerano due stringhe $x, y \in \Sigma^*$ tali che x sia in relazione $\mathscr{L}$ con y e si prenda una stringa $z \in \Sigma^*$:

$$x, y \in \Sigma^* : x \mathscr{R}_{\mathscr{L}} y, \quad z \in \Sigma^*$$

Supponendo che xz non sia in relazione $\mathscr{L}$ con yz, allora esiste una stringa w tale che xzw appartenga a $\mathscr{L}$ e yzw non appartenga a $\mathscr{L}$:

$$xz \not\mathscr{R}_{\mathscr{L}} yz \Rightarrow \exists w : x \underbrace{zw}_{u} \in \mathscr{L} \wedge y \underbrace{zw}_{u} \notin \mathscr{L}$$

Quindi implica l'esistenza di una stringa u appartenente a Σ^*, tale per cui xu appartiene a $\mathscr{L}$ e yu non appartiene a $\mathscr{L}$:

$$\Rightarrow \exists u \in \Sigma^* : xu \in \mathscr{L} \wedge yu \notin \mathscr{L}$$

Questo contraddice l'ipotesi (2.6): x e y non possono essere in relazione. Ne consegue che $x \not\mathscr{R}_{\mathscr{L}} y$, tuttavia avendo supposto per ipotesi $x \mathscr{R}_{\mathscr{L}} y$, risulta essere un assurdo e di conseguenza $\mathscr{R}_{\mathscr{L}}$ è invariante destra.

▶ $\mathscr{R}_M$ è invariante destra:

- Se partendo da q_0 (stato iniziale) si effettua un percorso x arrivando in q e successivamente tramite un altro percorso y si arriva ugualmente in q, allora $x \mathscr{R}_M y$ è vero. Questo significa che per ogni $w \in \Sigma^*$ si ha che $\hat{\delta}(q_0, xw)$ è uguale a q', di conseguenza anche con $\hat{\delta}(q_0, yw)$:

$$x \mathscr{R}_M y \Rightarrow \forall w \in \Sigma^* : \hat{\delta}(q_0, xw) = q' = \hat{\delta}(q_0, yw)$$

Quindi $\mathscr{R}_M$ è invariante destra.

2.4.2 Teorema di Myhill-Nerode

Teorema 2.4.4: Myhill-Nerode (1958)

I seguenti tre enunciati sono equivalenti:

1. $\mathscr{L} \subseteq \Sigma^*$ è regolare.
2. $\mathscr{L}$ è unione di classi di equivalenza su Σ^* indotte da una relazione invariante destra di indice finito.
3. $\mathscr{R}_{\mathscr{L}}$ è di indice finito.

Dimostrazione

Si dimostra la circolarità di equivalenza: $1 \Rightarrow 2 \Rightarrow 3 \Rightarrow 1$.

- **Prima implicazione** $1 \Rightarrow 2$:
  L'ipotesi è che $\mathscr{L} \subseteq \Sigma^*$ sia regolare. Allora $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ è un ASFD tale che $\mathscr{L} = \mathscr{L}(M)$ e pertanto $\mathscr{R}_M$ è invariante destra. Si ricorda inoltre che:
  $$x \mathscr{R}_M y \iff \hat{\delta}(q_0, x) = \hat{\delta}(q_0, y)$$
  ▷ Qual è la classe di equivalenza di una stringa x secondo la relazione $\mathscr{R}_M$? È l'insieme di tutte le $y \in \Sigma^*$ tale che x sia in relazione M con y:
  $$\begin{aligned} [x]_{\mathscr{R}_M} &= \{y \in \Sigma^* \mid x \mathscr{R}_M y\} \\ &= \{y \in \Sigma^* \mid \hat{\delta}(q_0, x)\} \\ &= \hat{\delta}(q_0, y) \end{aligned}$$
  Il linguaggio è l'unione per tutti i possibili stati q raggiungibili appartenenti a F dell'insieme di $y \in \Sigma^*$ tale che $\hat{\delta}(q_0, y) = q$:
  $$\mathscr{L} = \mathscr{L}(M) = \bigcup_{q \in F} \underbrace{\{y \in \Sigma^* \mid \hat{\delta}(q_0, y) = q\}}_{\text{Classe di equivalenza}}$$
  Quindi per tutte le stringhe che portano in uno stato finale, si uniscono le classi di equivalenza: $\mathscr{L}$ è l'unione di classi di equivalenza. Inoltre è una relazione invariante destra per il Lemma 2.4.1 ed è di indice finito poiché $|Q| < \omega$.
- **Seconda implicazione** $2 \Rightarrow 3$:
  Ogni relazione di equivalenza di indice finito e invariante destra, tale che $\mathscr{L}$ sia l'unione delle classi indotte dalla relazione (su Σ^*), è un raffinamento di $\mathscr{R}_{\mathscr{L}}$. Se ciò è vero, implica che $\mathscr{R}_{\mathscr{L}}$ sia la minima invariante destra, tale che $\mathscr{L}$ sia l'unione di classi di $\mathscr{R}_{\mathscr{L}}$.

Occorre dimostrare che $\mathscr{R}_{\mathscr{L}}$ ha indice finito, ovvero si partiziona l'insieme delle stringhe in un numero finito di classi d'equivalenza.

Se x appartiene alle stringhe, allora la classe di equivalenza secondo $\mathscr{R}$ è contenuta nella classe di equivalenza della stessa stringa x di $\mathscr{R}_{\mathscr{L}}$:

$$x \in \Sigma^* \Rightarrow [x]_{\mathscr{R}} \subseteq [x]_{\mathscr{R}_{\mathscr{L}}}$$

Questo significa che $\mathscr{R}$ raffina $\mathscr{R}_{\mathscr{L}}$.
Sia $y \in [x]_{\mathscr{R}}$, ovvero $x \mathscr{R} y$, ed essendo che per ipotesi $\mathscr{R}$ è invariante destra allora, per ogni $z \in \Sigma^*$, $xz \mathscr{R} yz$:

$$\forall z \in \Sigma^* : xz \mathscr{R} yz$$

Inoltre è stato supposto che $\mathscr{L}$ sia l'unione delle classi indotte dalla relazione $\mathscr{R}$, perciò $\mathscr{L}$ è:

$$\mathscr{L} = [x_1]_{\mathscr{R}} \cup \cdots \cup [x_n]_{\mathscr{R}}$$

dove $x_1, \ldots, x_n \in \Sigma^*$. Dentro una classe di equivalenza tutte le volte che ci sono due stringhe che appartengono alla stessa classe di equivalenza, cioè v è in relazione con w, implica che se v sta nel linguaggio allora anche w sta nel linguaggio:

$$v \mathscr{R} w \Rightarrow v \in \mathscr{L} \iff w \in \mathscr{L}$$

Dunque si prende una qualunque stringa $z \in \Sigma^*$ tale che xz appartenga a $\mathscr{L}$ se e solo se yz appartiene a $\mathscr{L}$:

$$\forall z \in \Sigma^* : xz \in \mathscr{L} \iff yz \in \mathscr{L}$$

Poiché $\mathscr{L}$ è l'unione di classi d'equivalenza, allora x è in relazione $\mathscr{L}$ con y, di conseguenza $y \in [x]_{\mathscr{R}_{\mathscr{L}}}$. Dato che l'indice di $\mathscr{R}_{\mathscr{L}}$ è minore uguale all'indice di $\mathscr{R}$ (finito per ipotesi), allora $\mathscr{R}_{\mathscr{L}}$ ha indice finito.

$$\texttt{ind}(\mathscr{R}_{\mathscr{L}}) \leq \texttt{ind}(\mathscr{R}) < \omega \Rightarrow \mathscr{R}_{\mathscr{L}} \text{ ha indice finito}$$

- **Terza implicazione** $3 \Rightarrow 1$:
  Si costruisce un ASFD, che riconosca $\mathscr{L}$, a partire dalla relazione $\mathscr{L}$ che possiede il minor numero di classi d'equivalenza.
  - Gli stati sono l'insieme delle classi di equivalenza di $\mathscr{R}_{\mathscr{L}}$, tale che x sia una qualunque stringa:

    $$Q' = \{[x]_{\mathscr{R}_{\mathscr{L}}} \mid x \in \Sigma^*\}$$

    Essendo per ipotesi che $\mathscr{R}_{\mathscr{L}}$ è di indice finito, allora la cardinalità di Q' è finita.
  - L'alfabeto è $\Sigma' = \Sigma$.

- Lo stato iniziale è la classe di equivalenza della stringa vuota:

$$q'_0 = [\varepsilon]_{\mathscr{R}_{\mathscr{L}}}$$

- La funzione di transizione di uno stato[a] leggendo un simbolo a corrisponde alla classe d'equivalenza xa secondo $\mathscr{R}_{\mathscr{L}}$:

$$\delta'([x]_{\mathscr{R}_{\mathscr{L}}}, a) = [xa]_{\mathscr{R}_{\mathscr{L}}}$$

- Gli stati finali sono l'insieme di tutte le classi di equivalenza delle stringhe, tali che x appartenga al linguaggio $\mathscr{L}$:

$$F' = \{[x]_{\mathscr{R}_{\mathscr{L}}} \mid x \in \mathscr{L}\}$$

Si dimostra che $\mathscr{L}(M) = \mathscr{L}$ con un'induzione sulle stringhe:

$$|y| \geq 0 : \hat{\delta}'([x]_{\mathscr{R}_{\mathscr{L}}}, y) = [x \cdot y]_{\mathscr{R}_{\mathscr{L}}} \tag{2.7}$$

- **Caso base**: se $|y| = 0$, allora $y = \varepsilon$.
  Infatti $\hat{\delta}'([x]_{\mathscr{R}_{\mathscr{L}}}, \varepsilon) = [x]_{\mathscr{R}_{\mathscr{L}}} = [x \cdot \varepsilon]_{\mathscr{R}_{\mathscr{L}}}$
- **Passo induttivo**: si suppone vera $\hat{\delta}'([x]_{\mathscr{R}_{\mathscr{L}}}, y) = [x \cdot y]_{\mathscr{R}_{\mathscr{L}}}$ per $|y| = n$ e si considera $|ya| = n + 1$.

$$\begin{aligned}\hat{\delta}'([x]_{\mathscr{R}_{\mathscr{L}}}, ya) &= \delta'(\hat{\delta}'([x]_{\mathscr{R}_{\mathscr{L}}}, y), a) \\ &\overset{\text{I.I.}}{=} \delta'([x \cdot y]_{\mathscr{R}_{\mathscr{L}}}, a) \\ &\overset{\text{def.}}{=} [x \cdot y \cdot a]_{\mathscr{R}_{\mathscr{L}}}\end{aligned}$$

  Allora (2.7) è dimostrata.

In funzione della proprietà per cui leggendo una stringa y in uno stato $[x]_{\mathscr{R}_{\mathscr{L}}}$ si ottiene uno stato $[x \cdot y]_{\mathscr{R}_{\mathscr{L}}}$, allora ne consegue che:

$$\begin{aligned}\hat{\delta}'(q'_0, x) &= \hat{\delta}'([\varepsilon]_{\mathscr{R}_{\mathscr{L}}}, x) \\ &= [\varepsilon x]_{\mathscr{R}_{\mathscr{L}}} = [x]_{\mathscr{R}_{\mathscr{L}}}\end{aligned}$$

In conclusione una stringa x appartiene a $\mathscr{L}(M')$ se e solo se si raggiunge uno stato finale:

$$\begin{aligned}x \in \mathscr{L}(M') &\iff \hat{\delta}'(q'_0, x) \in F' \\ &\iff [x]_{\mathscr{R}_{\mathscr{L}}} \in F' \\ &\iff x \in \mathscr{L}\end{aligned}$$

[a] In questo caso lo stato è una classe di equivalenza.

2.5 Proprietà fondamentali dei linguaggi regolari

Lemma 2.5.2: Pumping Lemma dei linguaggi regolari

Sia $\mathscr{L} \subseteq \Sigma^*$ regolare. Allora esiste un numero $n \in \mathbb{N}$ tale che, per ogni stringa $z \in \mathscr{L}$, se $|z| \geq n$ allora esistono $u, v, w \in \Sigma^*$ tali per cui:

1. La stringa z è suddivisa in uvw: $z = uvw$.
2. La lunghezza di uv è minore o uguale a n: $|uv| \leq n$.
3. La lunghezza di v è strettamente maggiore di 0: $|v| > 0$. Quindi in v è presente almeno un simbolo di Σ.
4. Per ogni i maggiore o uguale a 0, se si pompa v con i allora continua a essere parte di $\mathscr{L}$: $\forall i \geq 0. uv^i w \in \mathscr{L}$.

Questo lemma afferma che se si ha un linguaggio regolare, allora esiste sempre un numero finito n tale che, per ogni stringa più lunga di questo n, è possibile spezzare la stringa z in tre sottostringhe uvw, in modo che:

- Le prime due u e v hanno una lunghezza minore o uguale a n.
- Dentro v c'è almeno un simbolo.
- Si può ripetere la sottostringa v un numero arbitrario di volte e rimanere ancora in $\mathscr{L}$.

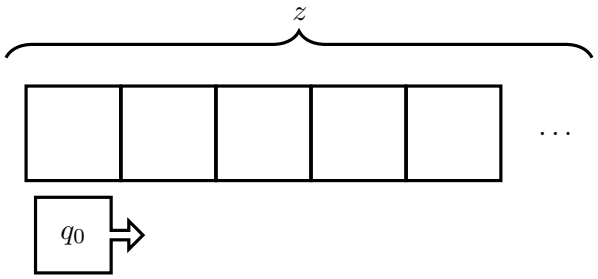
Dimostrazione

L'ipotesi di partenza è che $\mathscr{L}$ sia regolare: $\mathscr{L} \in \text{REG}$. Ciò implica per definizione che esista un ASFD M tale che $\mathscr{L} = \mathscr{L}(M)$ e M sia l'automa minimo (ricavato dal teorema di Myhill-Nerode). La quintupla dell'automa è definita come:

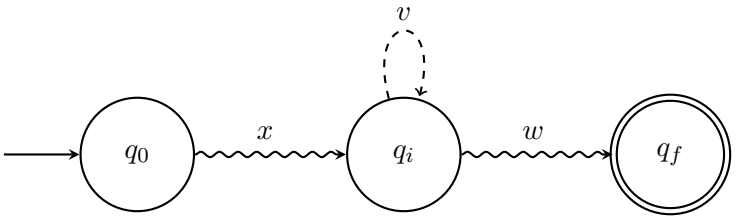
$$M = \langle Q, \Sigma, \delta, q_0, F \rangle$$

Occorre dimostrare le quattro affermazioni del lemma:

1. Il numero di stati Q è finito e sia $k = |Q| < \omega$; esiste sicuramente un numero $n > k$, come $n = k + 1$. Sia la stringa $z \in \Sigma^*$ tale che $z \in \mathscr{L}$ e $|z| \geq n$, questo implica che $|z| > |Q| = k$.



Quindi la macchina ha sul nastro una stringa z più lunga del numero di stati. Parte dallo stato q_0 e a ogni lettura effettua una transizione di stato. Poiché $|z|$ è maggiore del numero di stati, allora sicuramente qualche stato viene ripetuto.



Si sceglie q_i come stato che si incontra più di una volta leggendo z. La stringa z viene suddivisa come:

- Sottostringa u: porta da q_0 a toccare q_i per la prima volta.
- Sottostringa v: porta a rincontrare q_i almeno una volta.
- Sottostringa w: porta da q_i a q_f.

Da questo deriva che $z = uvw$.

2. Essendo q_i il primo stato che si incontra più di una volta, allora non può essere che $|uv| > n$, perché altrimenti la porzione di stringa $|uv|$ diventerebbe più lunga del numero di stati e ci sarebbe un altro stato q_j che si incontrerebbe più di una volta prima di q_i. Di conseguenza $|uv| \leq n$.

3. Per incontrare più di una volta lo stato q_i è necessario $|v| > 0$. In caso contrario lo stato non verrebbe incontrato più di una volta.

4. Dal grafo è visibile che con la sottostringa u si passa da q_0 a q_i, con v^i si continua a tornare su q_i e infine con w si passa da q_i a q_f. Di conseguenza $\forall i \geq 0 : uv^iw \in \mathscr{L}$.

Osservazione

Nella dimostrazione è stato preso un ASFD, ma non cambierebbe nulla scegliere un ASFND o ε-ASFND, perché sono entrambi riducibili a un ASFD e successivamente minimizzabile.

La macchina intesa come automi a stati finiti ha una memoria finita ed è limitata dal numero di stati, pertanto non si riesce a ricordare il numero di iterazioni per un certo stato.

Dato questo lemma, valgono le due seguenti proprietà:

1. Se $\mathscr{L}$ è regolare, allora vale il *pumping* lemma linguaggi regolari:

$$\mathscr{L} \in \text{REG} \Rightarrow \text{PL linguaggi regolari}$$

Quindi esiste un numero $n \in \mathbb{N}$ tale che per ogni z appartenente alle stringhe Σ^*, se $z \in \mathscr{L}$ e $|z| \geq n$, allora esiste la stringa uvw appartenente alle stringhe Σ^* tale per cui $z = uvw$, $|uv| \leq n$, $|v| > 0$ e $\forall i \in \mathbb{N} : uv^i w \in \mathscr{L}$:

$$\exists n \in \mathbb{N}. \forall z \in \Sigma^* : z \in \mathscr{L} \wedge |z| \geq n \Rightarrow \exists u, v, w \in \Sigma^* : \underbrace{z = uvw}_{\mathbf{c_1}} \wedge \underbrace{|uv| \leq n}_{\mathbf{c_2}} \wedge \underbrace{|v| > 0}_{\mathbf{c_3}} \wedge \underbrace{\forall i \in \mathbb{N} : uv^i w \in \mathscr{L}}_{\mathbf{c_4}}$$

2. Se si nega il PL linguaggi regolari, allora $\mathscr{L}$ non è regolare:

$$\neg\text{PL linguaggi regolari} \Rightarrow \mathscr{L} \notin \text{REG}$$

Si nega l'espressione vista per la prima proprietà:

$$\begin{aligned} \forall n \in \mathbb{N}. \exists z \in \Sigma^* : z \in \mathscr{L} \wedge |z| \geq n \wedge \forall u, v, w \in \Sigma^* &: \neg(\mathbf{c_1} \wedge \mathbf{c_2} \wedge \mathbf{c_3} \wedge \mathbf{c_4}) \\ &: (\mathbf{c_1} \wedge \mathbf{c_2} \wedge \mathbf{c_3}) \vee \neg \mathbf{c_4} \\ &: (\mathbf{c_1} \wedge \mathbf{c_2} \wedge \mathbf{c_3}) \Rightarrow \neg \mathbf{c_4} \\ &: (z = uvw \wedge |uv| \leq n \wedge |v| > 0) \Rightarrow \exists i : uv^i w \notin \mathscr{L} \end{aligned}$$

Si ricorda che $A \Rightarrow B$ corrisponde a scrivere $\neg A \vee B$ e viceversa.

Quindi per dimostrare che un linguaggio non sia regolare:

- si suppone un numero n generico;

- si sceglie una stringa z dipendente da n e presente nel linguaggio;
- infine si dimostra che per ogni possibile suddivisione della stringa, in tre sottostringhe uvw, esiste un valore i tale per cui pompando/spompando v si esce dal linguaggio $\mathscr{L}$.

Esempio

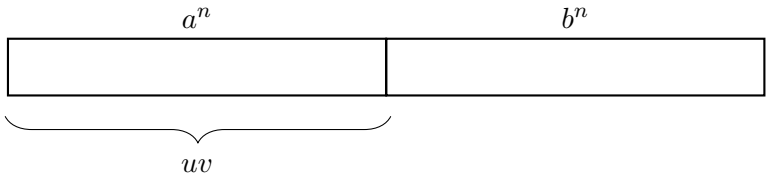
Si considera il linguaggio $\mathscr{L} = \{a^n b^n \mid n \geq 0\}$ e per un qualunque n si deve decidere se la stringa $a^n b^n \in \mathscr{L}$. Tuttavia se l'unica memoria a disposizione è lo stato, allora per ricordare la precisa quantità di a e b incontrati occorre uno stato per ciascun simbolo, ma in questo modo diventerebbe un automa a stati infiniti.

$\triangleright$ $\mathscr{L}$ è regolare?

- Si prende un generico $n \in \mathbb{N}$ e si considera una stringa $a^n b^n$ dipendente da n e lunga maggiore o uguale di n:

$$a^n b^n \in \mathscr{L}, \quad |a^n b^n| \geq n$$

- Per ogni possibile suddivisione in tre stringhe si deve avere:
  - $|uv| \leq n$, infatti:



    uv non contiene elementi di b^n, ma solo: $uv = a^k$ con $k \leq n$.
  - $|v| > 0$, infatti esiste un valore i per cui esiste un pompaggio che esce da $\mathscr{L}$:

$$i = 2 \quad \rightarrow \quad uv^2w = uvvw$$

    Poiché $uv = a^k$, allora implica che $v = a^h$ con $h \leq n$. Pertanto nella stringa pompata il numero di a è strettamente maggiore del numero di b e $uv^2w \notin \mathscr{L}$.

Ecco dimostrata la negazione del *pumping* lemma:

$$\forall n \in \mathbb{N}. \exists z \in \Sigma^* : z \in \mathscr{L} \wedge |z| \geq n \wedge \forall u, v, w \in \Sigma^* : \begin{array}{l} z = uvw \ \wedge \ |uv| \leq n \wedge \\ |v| > 0 \Rightarrow \exists i : uv^i w \notin \mathscr{L} \end{array}$$

Poiché è dimostrata la negazione del *pumping* lemma, allora $\mathscr{L} \notin$ REG.

Esempio

Si considera il linguaggio $\mathscr{L} = \{a^{n^2} \mid n \geq 1\}$ e per un qualunque n si deve decidere se $a^{n^2} \in \mathscr{L}$. Tuttavia non si riesce ad avere abbastanza memoria, perché è proporzionale alla crescita del numero.

$\triangleright$ $\mathscr{L}$ è regolare?

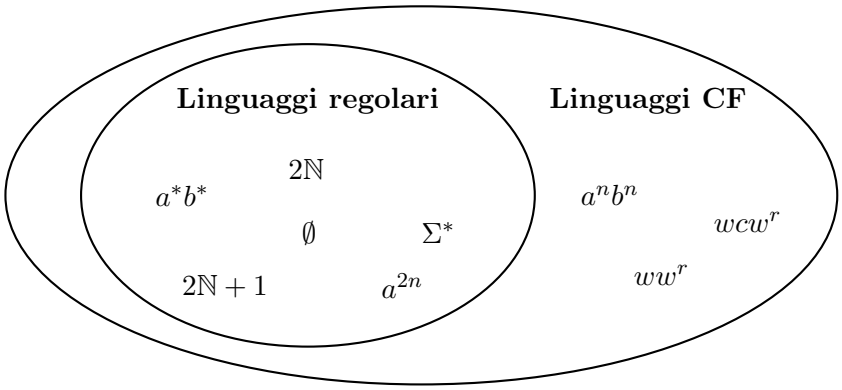
- Si prende un generico $n \in \mathbb{N}$ e una stringa $z = a^{n^2}$ dipendente da n e lunga maggiore o ugual di n:
$$a^{n^2} \in \mathscr{L}, \quad \left|a^{n^2}\right| \geq n$$
- Per ogni possibile suddivisione in tre stringhe si deve avere $|uv| \leq n$ e un valore i tale che il pompaggio di v esca da $\mathscr{L}$. Sapendo che:
$$|uv| \leq n \wedge |v| \geq 1 > 0 \quad \rightarrow \quad 1 \leq v \leq n$$
Si sceglie $i = 2$, così da ricavare:
$$|z| = n^2 < |uv^2w| = |uvvw| = |\underbrace{uvw}_{n^2} v| \leq n^2 + n < (n+1)^2$$
La stringa pompata uv^2w è compresa tra n^2 e $(n+1)^2$, di conseguenza la lunghezza di uv^2w non è un quadrato. In conclusione $uv^2w \notin \mathscr{L}$.

Poiché è dimostrata la negazione del *pumping* lemma, allora $\mathscr{L} \notin$ REG.

Automi a pila

3.1 Grammatiche libere da contesto

Le **grammatiche libere da contesto** riguardano ancora i **linguaggi formali** e permettono di trattare i linguaggi che gli ASFD, ASFND, ε-ASFND non possono accettare.



▷ Se questa è l'attuale situazione, come si può costruire una macchina in grado di risolvere problemi decisionali legati a linguaggi non regolari?
Questa nuova famiglia di linguaggi è chiamata **linguaggi liberi da contesto** (*context free*, CF) e include i linguaggi regolari come caso particolare. Nel dettaglio consente di trattare i linguaggi di programmazione: ogni descrizione linguistica di un linguaggio di programmazione possiede una grammatica in BNF[1] (*Backus-Naur form*).

Un linguaggio di programmazione deve contenere almeno il sistema delle parentesi[2] e di conseguenza non può essere riconosciuto da un linguaggio naturale. L'algoritmo che decide se il programma scritto è corretto, in un qualsiasi linguaggio di programmazione, è noto come ***parser***. La fase di *parsing* è la prima fase di compilazione ed è determinata dal concetto di linguaggio libero da contesto.

[1] All'interno del BNF è definita la struttura (la forma) della grammatica per un linguaggio di programmazione.

[2] Tutte le volte che si aprono delle parentesi, successivamente vanno chiuse in egual numero.

Storicamente i linguaggi liberi da contesto nascono dalle analisi di Chomsky.

3.1.1 Grammatica generativa

Una grammatica **generativa**, ossia che genera stringhe, secondo Chomsky, è una quadrupla composta da:

$$G = \langle V, T, P, S \rangle$$

- V è un insieme finito di variabili **non-terminali**.
- T è un insieme finito di caratteri **terminali**, ovvero corrispondente all'alfabeto Σ.
- P è un insieme finito di **regole di riscrittura** o **produzioni**.
- $S \in V$ è il simbolo iniziale appartenente all'insieme di simboli non-terminali.

Per grammatiche CF (libere da contesto) le produzioni sono di questo tipo:

$$A \to \alpha \in P, \quad \text{dove } A \in V \text{ e } \alpha \in (T \cup V)^*$$

Quindi A appartiene a un insieme finito di variabili non-terminali e α appartiene a una qualunque sequenza di terminali e non-terminali. L'unico vincolo è che non esistono vincoli: quello che si ottiene rimpiazzando A con α è una stringa che contiene variabili terminali o non-terminali.

▷ Cosa significa "essere generativo" in una grammatica libera da contesto?
Si vede il concetto di **linguaggio generato** da:

$$G = \langle V, T, P, S \rangle$$

Per definire questo linguaggio occorre vedere cosa si intende per "generare una stringa". Se una stringa contiene al suo interno dei non-terminali, allora si possono rimpiazzare queste variabili con quello che c'è a destra di una produzione.

- $A \to \beta$ è una produzione P e α, γ sono sequenze di terminali e non-terminali qualunque:

  $$A \to \beta \wedge \alpha, \gamma \in (V \cup T)^*$$

  Se si dispone di una stringa A, preceduta da una sequenza α e seguita da γ, allora secondo la grammatica G, applicando la produzione $A \to \beta \in P$, questa diventa $\alpha\beta\gamma$ dove β rimpiazza A:

  $$\alpha A \gamma \xrightarrow{G} \alpha\beta\gamma$$

  Questo è un **passo di derivazione** o **riscrittura**: si riscrive $\alpha A \gamma$ in $\alpha\beta\gamma$; è un passo in cui una variabile viene rimpiazzata con il contenuto della produzione a destra di quella variabile.

▷ Perché è libero da contesto?
Perché a prescindere da cosa siano α e γ, che stanno a sinistra e a destra della variabile, tutte le volte che è presente un simbolo non-terminale A lo si può rimpiazzare con la parte destra della produzione in cui A è parte sinistra. Quindi non si guarda cosa sta attorno al simbolo non-terminale, ovvero il suo contesto.

- Se $\alpha_1, \ldots, \alpha_i$ sono sequenze di simboli terminali e non-terminali, con $i \geq 1$, e per ogni $j = 1, \ldots, i-1$ si ha che α_j va in α_{j+1} con un passo:

$$\alpha_1, \ldots, \alpha_{i \geq 1} \in (V \cup T)^* \wedge \forall j = 1 \ldots i-1 : \alpha_j \overset{G}{\longleftrightarrow} \alpha_{j+1}$$

  Allora una **sequenza di derivazione** consiste in:

$$\alpha_1 \overset{G}{\longrightarrow} \alpha_2 \overset{G}{\longrightarrow} \alpha_3 \overset{G}{\longrightarrow} \ldots \overset{G}{\longrightarrow} \alpha_i$$
$$\alpha_1 \overset{G}{\longrightarrow}^{i} \alpha_i$$

- Se esiste un numero n tale che α_1 in G e con n passi va in una certa configurazione α_n, allora α_1 si indica con $*$ gli n passi:

$$\exists n \in \mathbb{N} : \alpha_1 \overset{G}{\longrightarrow}^{n} \alpha_n \equiv \alpha_1 \overset{G}{\longrightarrow}^{*} \alpha_n$$

- G sia una grammatica CF, allora il linguaggio generato da G è uguale all'insieme delle stringhe sui soli terminali, tale che dal simbolo iniziale con un certo numero di passi si ottenga la stringa σ:

$$\mathscr{L}(G) = \{\sigma \in T^* \mid S \overset{G}{\longrightarrow}^{*} \sigma\}$$

Quando si parla di grammatiche: il linguaggio è quello generato dalla grammatica: tutte le stringhe possono essere generate con questo processo di riscrittura a partire dal simbolo iniziale S (variabile singola), applicando successivamente la riscrittura di un non-terminale con la stringa corrispondente di ogni produzione, fintantoché non si raggiunge una stringa di soli terminali.

Esempio

Si considera un linguaggio $\mathscr{L} = \{a^n b^n \mid n \geq 0\}$. Questo linguaggio non è regolare, ma è libero da contesto.

▷ Come si costruisce una grammatica?
La grammatica segue la quadrupla $G = \langle V, T, P, S \rangle$ per cui:

- V (non-terminali) contiene il simbolo $\{S\}$.
- T (terminali) contiene $\{a, b\}$.
- $S \in V$ è il simbolo iniziale appartenente ai non-terminali.

- Per costruire P si ragione induttivamente:
  - **Caso base**: se $\varepsilon \in \mathscr{L}$, allora $S \to \varepsilon$.
  - **Passo induttivo**: se S genera con un certo numero di passi $a^n b^n$ (corrispondente a S), cosa accade se si aggiunge a sinistra il simbolo a e a destra il simbolo b?
    aSb è uguale per esteso a $a \cdot a^n b^n \cdot b$, di conseguenza si ottiene:

    $$S \xrightarrow{G}{}^{*} a^{n+1} b^{n+1} \in \mathscr{L}$$

    Il caso $n+1$ continua a restare in $\mathscr{L}$.

In conclusione la grammatica di questo linguaggio corrisponde a:

$$\begin{cases} S \to \varepsilon \\ S \to aSb \end{cases}$$

Nella notazione BNF si scrive $S \to \varepsilon \,|\, aSb$.

Non serve ogni volta indicare la quadrupla, basta fornire soltanto le produzioni.

Dimostrazione: Uguaglianza di $\mathscr{L}(G)$

Bisogna dimostrare le due inclusioni $\subseteq$ e $\supseteq$ dell'uguaglianza:

$$\mathscr{L}(G) = \underbrace{\{a^n b^n \mid n \geq 0\}}_{\mathscr{L}}$$

- **Prima inclusione $\mathscr{L} \subseteq \mathscr{L}(G)$**: se $x \in \mathscr{L}$, allora $x \in \mathscr{L}(G)$. Si effettua l'induzione sulle stringhe in $\mathscr{L}$, le quali sono tutte di lunghezza pari:
  - **Caso base**: se $n = 0$, allora $x = \varepsilon$ e si ha come produzione che da S si va in ε:

    $$S \to \varepsilon$$

    Ciò implica che $\varepsilon \in \mathscr{L}(G)$.
  - **Passo induttivo**: se $n > 0$, allora per ipotesi induttiva si sa che $a^n b^n \in \mathscr{L}$ e da S con un certo numero di passi si produce $a^n b^n$:

    $$a^n b^n \in \mathscr{L}, \quad S \xrightarrow{G}{}^{*} a^n b^n$$

    Il caso successivo a $n+1$ comporta la stringa $a^{n+1} b^{n+1}$, ossia $a a^n b^n b$. Si verifica se esiste una derivazione sulla grammatica:

  - da S si applica la prima produzione e si va in aSb;
  - per ipotesi si ha che $a^n b^n$ è generata dalla grammatica e di conseguenza si va da aSb ad $aa^n b^n b$ con un certo numero di passi in G:
$$S \to aSb \xrightarrow{G}{}^{*} aa^n b^n b = a^{n+1} b^{n+1}$$

- **Seconda inclusione $\mathscr{L}(G) \subseteq \mathscr{L}$**: se $x \in \mathscr{L}(G)$, allora $x \in \mathscr{L}$. Poiché in $\mathscr{L}(G)$ ci sono infinite stringhe che si possono enumerare, secondo la lunghezza della derivazione per generarle, allora si effettua un'induzione sulla lunghezza della derivazione.

  - **Caso base**: se $n = 1$, allora l'unica derivazione che porta solo terminali di lunghezza unitaria è:
$$S \xrightarrow{G} \varepsilon$$
Infatti $\varepsilon \in \mathscr{L}$.

    Il caso base non può partire da $n = 0$, perché con una derivazione lunga 0 rimane un simbolo non-terminale e una stringa appartiene al linguaggio generato se è una stringa con soli simboli terminali.

  - **Passo induttivo**: se $n > 1$, allora per ipotesi induttiva per ogni m minore o uguale a n si ha che S, con un certo numero di passi m, porta a x. Ciò implica che x appartenga a $\mathscr{L}$.
$$\forall m \leq n : S \xrightarrow{G}{}^{m} x \Rightarrow x \in \mathscr{L}$$
Quindi sempre per ipotesi induttiva si ha:
$$S \xrightarrow{G} aSb \xrightarrow{G}{}^{n} axb, \quad x \in \mathscr{L}$$
Si prende una derivazione più lunga di un passo: se è noto che $x \in \mathscr{L}$, allora $x = a^h b^h$, pertanto si considera $aa^h b^h b$ che è uguale a $a^{h+1} b^{h+1}$. Ovviamente $a^{h+1} b^{h+1} \in \mathscr{L}$.

3.1.2 Alberi di derivazione

Un **albero di derivazione** ha lo scopo di rappresentare sequenze di derivazione che sono diverse. Prendendo il seguente linguaggio:

$$G \begin{cases} S \to ASB \mid \varepsilon \\ A \to a \\ B \to b \end{cases}$$

La stringa $aabb \in \mathscr{L}(G)$ ha due sequenze di derivazione differenti in questa grammatica:

- Prima possibile risoluzione:

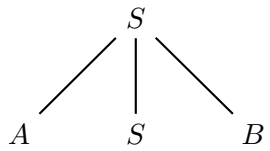
$$S \to ASB \to AASBB \to AABB \to aABB \to aaBB \to aabB \to aabb$$

- Seconda possibile risoluzione:

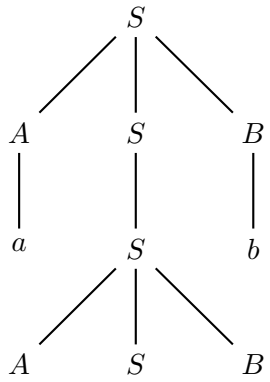
$$S \to ASB \to AASBB \to aASBB \to aaSBB \to aaSBb \to aaBb \to aabb$$

Durante una derivazione non è obbligatorio seguire uno specifico ordine; ogni volta si può avere una sequenza diversa.

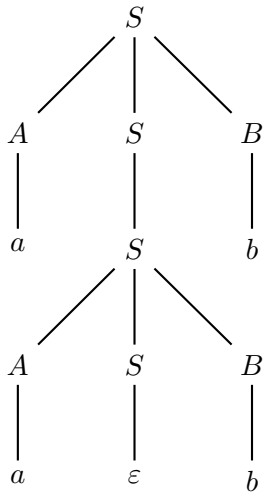
L'**albero di derivazione** ha come radice il simbolo iniziale e tutte le volte in cui si ha una produzione $S \to ASB$ si hanno tre figli:



A questo punto A e B possono avere un solo figlio, mentre S può avere nuovamente A, S e B:



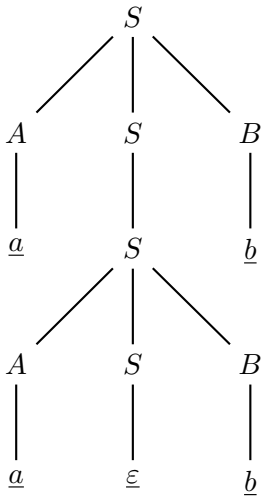
In questo caso possono avere solo un figlio:



Osservazione

I simboli alla frontiera sono detti "terminali" perché corrispondono alle foglie dell'albero, mentre i nodi interni sono le variabili che possono essere sostitute.

Una visita di questo albero che enumera tutti i terminali generati dall'albero restituisce esattamente la stringa. Infatti:

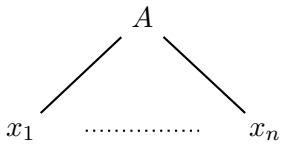


$aabb \in \mathscr{L}(G)$

- Le foglie sono simboli in T.
- I nodi sono le variabili in V.

- La radice è $S \in V$, che è il simbolo iniziale.

Se si dispone di $A \to x_1, \ldots, x_n \in P$, allora si genera l'albero:



3.1.3 Grammatiche ambigue

Si considera la grammatica delle espressioni numeriche EXP (simbolo iniziale), che può essere un numero, una somma di due espressioni numeriche, il prodotto di due espressioni numeriche oppure può essere un'espressione numerica tra parentesi:

$$\text{EXP} \to \text{NUM} \,|\, \text{EXP} + \text{EXP} \,|\, \text{EXP} * \text{EXP} \,|\, (\text{EXP})$$

Un numero NUM può essere una cifra oppure una cifra seguita da un numero:

$$\text{NUM} \to \text{CIF} \,|\, \text{CIFNUM}$$

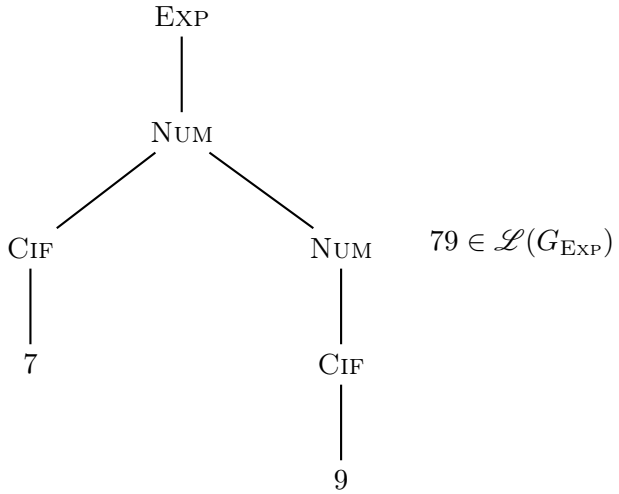
Una cifra può essere un valore da 0 a 9:

$$\text{CIF} \to 0 \,|\, \ldots \,|\, 9$$

Quindi per le espressioni questa è una grammatica:

$$G_{\text{EXP}} \begin{cases} \text{EXP} \to \text{NUM} \,|\, \text{EXP} + \text{EXP} \,|\, \text{EXP} * \text{EXP} \,|\, (\text{EXP}) \\ \text{NUM} \to \text{CIF} \,|\, \text{CIFNUM} \\ \text{CIF} \to 0 \,|\, \ldots \,|\, 9 \end{cases}$$

Se bisogna scrivere il numero 79, l'albero di derivazione diventa:

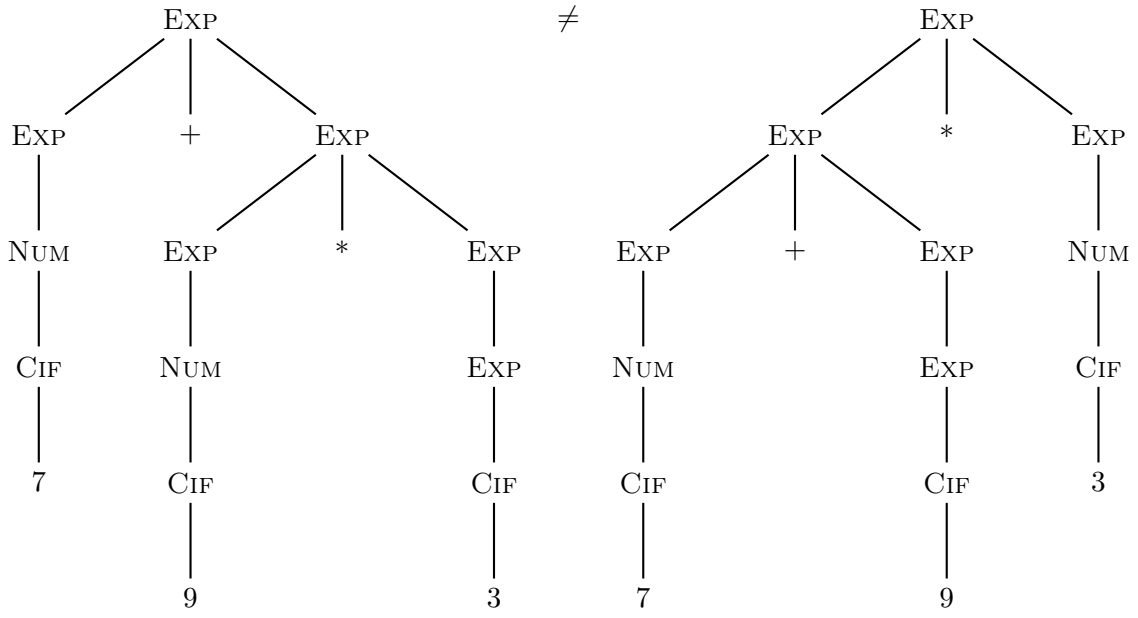


Ogni volta che esiste un albero di derivazione, avente come radice il simbolo iniziale, per una stringa nella grammatica, allora la stringa appartiene al linguaggio generato dalla grammatica e viceversa.

Tuttavia gli alberi di derivazione possono essere diversi anche per la stessa stringa: mentre un albero di derivazione rappresenta più sequenze di derivazione, ci possono essere per una stringa più di un albero di derivazione.

Esempio

La stringa $7 + 9 * 3 \in \mathscr{L}(G_{\text{Exp}})$?
Ci sono due possibili alberi di derivazione che però non sono uguali:



In questo caso la stringa $7 + 9 * 3 \in \mathscr{L}(G_{\text{Exp}})$ per entrambi gli alberi; basta soltanto trovare un albero corretto. Nel momento in cui esiste più di un albero per una stringa, allora la grammatica si dice **ambigua**: G_{Exp} è ambigua.

Legata all'ambiguità ci può essere un'**interpretazione**. Per esempio se una calcolatrice dovesse interpretare, con una visita in profondità[3], la successione delle operazioni da eseguire, allora ci sarebbero dei problemi nel risultato dell'operazione. Infatti, seguendo gli alberi ricavati nell'esempio precedente, l'albero a sinistra esegue prima la moltiplicazione (operazione più bassa) e poi la somma (operazione più alta) dando come risultato 34, mentre l'albero a destra esegue prima la somma (operazione più bassa) e successivamente il prodotto (operazione più alta) restituendo 48.

[3]Parte dalle operazioni in basso per poi risalire.

Un modo per eliminare l'ambiguità è quello di forzare determinate operazioni a restare più basse nell'albero.

Teorema 3.1.1: Corrispondenza grammatica-albero

Sia G una grammatica libera da contesto. Se da S, con una grammatica G e un certo numero di passi, si arriva alla stringa α, allora ciò è vero se e solo se esiste un albero di derivazione con radice S e foglie α.

$$S \xrightarrow{G}{}^{*} \alpha \iff \exists \overset{S}{\underset{\alpha}{\Delta}}$$

Dimostrazione

Si dimostra che per ogni non-terminale A succede che: se da A, con una grammatica G e un certo numero di passi, si raggiunge la stringa α, allora questo è vero se e solo se esiste un albero di radice A e base α:

$$\forall A \in V : A \xrightarrow{G}{}^{*} \alpha \iff \exists \overset{A}{\underset{\alpha}{\Delta}}$$

Si effettua un'induzione sulla lunghezza del numero di passi della derivazione:

- **Caso base**: se $n = 0$, allora da A con 0 passi si arriva in α se e solo se $A = A$ e l'albero è un solo punto (A):

$$A \xrightarrow{G}{}^{0} \alpha \iff A = A \qquad \bullet^{A}$$

- **Passo induttivo**: se $n > 0$, allora si ha come ipotesi induttiva che per ogni A appartenente a V si ottiene che da A, con una grammatica G e n passi, si genera α se e solo se esiste l'albero con radice A e base α:

$$\forall A \in V : A \xrightarrow{G}{}^{n} \alpha \iff \overset{A}{\underset{\alpha}{\Delta}}$$

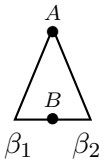
  Quindi si suppone una radice $A \in V$ e che da A, con una grammatica G e n passi, si genera $\beta_1 B \beta_2$ con $B \in V$:

$$A \in V,\ A \xrightarrow{G}{}^{\overbrace{n}^{\text{Ipotesi induttiva}}} \underbrace{\beta_1 B \beta_2}_{\alpha}, \quad \text{con } B \in V$$

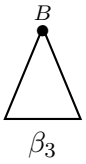
Inoltre $B \to \beta_3$ appartiene alle produzioni, di conseguenza si deriva che da A con n passi si va in $\beta_1 B \beta_2$, dopodiché tramite un passo in $\beta_1\beta_3\beta_2$:

$$\underbrace{A \xrightarrow{G}{}^{n} \beta_1 B \beta_2 \xrightarrow{G} \beta_1\beta_3\beta_2}_{n+1 \text{ passi}}$$

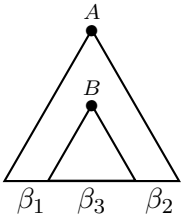
Per ipotesi induttiva è noto che c'è un albero che possiede la radice A e la frontiera $\beta_1 B \beta_2$:



Sapendo poi che $B \to \beta_3 \in P$ nella grammatica G, allora esiste un albero con radice B e base β_3:



Infine è possibile unire quest'ultimo albero all'altro, essendo libero da contesto, così da generare:



3.2 Forma normale per i linguaggi CF

Si prende la seguente grammatica:

$$\begin{cases} S \to aSb \mid \varepsilon \mid H \\ H \to aH \\ N \to bN \mid b \end{cases}$$

Da S si può generare H e a sua volta da H si può generare aH. Tuttavia non c'è nessun modo per chiudere H:

$$aH \to aaH \to aaaH \to \dots$$

Quindi da questa H non si riesce a produrre una sequenza di soli terminali: ci sarà sempre in mezzo la variabile H. Al contrario da N è permesso generare sequenze di soli terminali:

$$N \to bN \to bbN \to bbbN \to bbbb$$

Però N non è raggiungibile in alcun modo dal simbolo iniziale S.

Da questa grammatica è possibile rimuovere dei **simboli inutili** ai fini delle stringhe generabili.

3.2.1 Simboli utili e inutili

Vengono definiti due punti:

1. Un simbolo $x \in T$ (terminale) oppure $x \in V$ (non-terminale) è **utile** se compare in almeno una derivazione per qualche stringa appartenente ai soli terminali $\sigma \in T^*$ in $\mathscr{L}$.

2. Non devono essere presenti produzioni inutili, come $A \to B$, dette **unitarie**. Una produzione è unitaria se riscrive una variabile con un'altra.

Partendo dal primo punto, sia G una grammatica CF. Un simbolo $x \in T$ è **utile** se esiste $\sigma \in T^*$ tale che da S con una sequenza di produzioni nella grammatica G si arriva fino a una sequenza $\alpha x \beta$:

$$S \xrightarrow{G}{}^{*} \alpha x \beta$$

Dove α, β possono essere qualunque sequenze di terminali/non-terminali e a sua volta questa stringa porta a σ:

$$S \xrightarrow{G}{}^{*} \alpha x \beta \xrightarrow{G}{}^{*} \sigma$$

Il che vuol dire in modo sottinteso che $\sigma \in \mathscr{L}(G)$. Un simbolo $x \in V \cup T$ è **inutile** se non è utile.

3.2.2 Eliminazione dei simboli inutili

Per eliminare i **simboli inutili** vengono impiegati i seguenti lemmi tenendo come esempio la precedente grammatica:

$$\begin{cases} S \to aSb \mid \varepsilon \mid H \\ H \to aH \\ N \to bN \mid b \end{cases}$$

Lemma 3.2.1: Eliminazione delle variabili che non producono terminali

Sia $G = \langle V, T, P, S\rangle$ una grammatica CF tale che il linguaggio generato da G non sia vuoto $\mathscr{L}(G) \neq \emptyset$. Allora esiste $G' = \langle V', T, P', S\rangle$ una grammatica CF tale che per ogni $A \in V'$ esiste una stringa di soli terminali in T', tale che A nella grammatica G' produce la stringa σ di terminali e $\mathscr{L}(G) = \mathscr{L}(G')$:

$$\forall A \in V'.\, \exists \sigma \in T^* : A \xrightarrow{G'}{}^{*} \sigma \wedge \mathscr{L}(G) = \mathscr{L}(G')$$

Osservazione

Si mantengono T e S invariati, poiché:

- T sono i terminali non prodotti dalle variabili da eliminare. Pertanto non ha senso di modificare questo insieme.
- S è l'insieme da cui si produce una stringa di terminali. Dunque creare un S' per poi eliminarlo (se non produce terminali) significherebbe avere un linguaggio vuoto, cioè impossibile perché $\mathscr{L}(G) \neq \emptyset$.

Dimostrazione

Si costruisce la grammatica G' costruendo gli insiemi V' e P' (unici insiemi modificabili rispetto G).

- Si costruisce l'operatore iteratore Γ che va dalle parti di V nelle parti di V:

  $$\Gamma : 2^V \to 2^V$$

  Per ogni $W \subseteq V$, $\Gamma(W)$ è l'insieme di tutti i non-terminali, tale per cui esiste una stringa α di terminali/non-terminali tale che da A in α appartenga a P:

  $$\forall W \subseteq V, \quad \Gamma(W) = \{A \in V \mid \exists \alpha \in (T \cup W)^*.\, A \to \alpha \in P\}$$

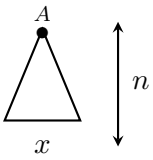
Da questa costruzione si osserva che:

1. Γ è **monotona**: se $W_1 \subseteq W_2$ allora $\Gamma(W_1) \subseteq \Gamma(W_2)$.
2. Le iterazioni sono definite in modo induttivo:

   $$\begin{cases} \Gamma^0(W) = W \\ \Gamma^{n+1}(W) = \Gamma(\Gamma^n(W)) \end{cases}$$

Per induzione su $n \in \mathbb{N}$, un simbolo non-terminale appartiene all'iterazione n-esima a partire dall'insieme vuoto se e solo se esiste una stringa x appartenente ai soli terminali, tale che da A, con un albero, si raggiunge x dove l'altezza è proporzionale a n:

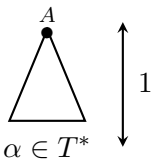
$$A \in \Gamma^n(\emptyset) \iff \exists x \in T^* :$$



- **Caso base**: per $n = 1$ si ha che:

$$A \in \Gamma'(\emptyset) = \Gamma(\emptyset) = \{A \in V \mid \exists \alpha \in T^*. A \to \alpha \in P\}$$

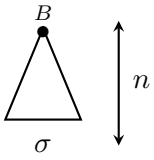
Questo è vero se e solo se:



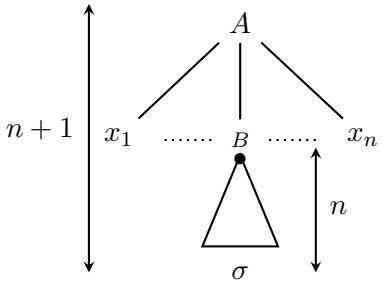
- **Passo induttivo**: per $n > 1$ si ha come ipotesi induttiva che:

$$\begin{aligned} A \in \Gamma(\Gamma^n(\emptyset)) &= \{A \in V \mid \exists \alpha.(T \cup \Gamma^n(\emptyset))^*. A \to \alpha \in P\} \\ &\iff A \to x_1 \dots B \dots x_n, \quad B \in \Gamma^n(\emptyset) \wedge x_i \in T \end{aligned}$$

$$\iff \exists \sigma \in T^* :$$



Quindi per un'altezza $n+1$ si ricava che da A si va in $x_1 \dots B \dots x_n$ e dopodiché viene applicata l'ipotesi induttiva:



Poiché Γ è monotona, allora $|V| < \omega$ e di conseguenza $\left|2^V\right| < \omega$:

$$|V| < \omega \Rightarrow \left|2^V\right| < \omega$$

Ciò vuol dire che aggiungendo simboli non-terminali non si può crescere all'infinito. A un certo punto è obbligatorio fermarsi, perché esiste un limite massimo, ossia l'insieme di tutti i non-terminali, oltre al quale non si può aggiungere altri non-terminali. Infatti se si considera per esempio $\emptyset \subseteq \Gamma(\emptyset)$, allora si può applicare Γ in entrambi i lati:

$$\begin{gathered}\emptyset \subseteq \Gamma(\emptyset) \\ \Gamma(\emptyset) \subseteq \Gamma(\Gamma(\emptyset)) \\ \Gamma(\Gamma(\emptyset)) \subseteq \Gamma(\Gamma(\Gamma(\emptyset))) \\ \vdots\end{gathered}$$

Questo contenimento non può continuare all'infinito, perché gli insiemi delle variabili e dei terminali sono finiti. Pertanto esiste un numero massimo di iterazioni oltre al quale Γ si stabilizza al cosiddetto **punto fisso** (*fix point*), cioè all'insieme per cui l'iterazione non altera l'insieme.
Si ricava che, per ogni n appartenente ai numeri naturali, $\Gamma^n(\emptyset)$ è contenuta nella massima iterazione:

$$\forall n \in \mathbb{N}, \quad \Gamma^n(\emptyset) \subseteq \underbrace{\Gamma^{|V|}(\emptyset)}_{\text{punto fisso}}$$

ciò implica che applicando ancora una volta Γ si ha che:

$$\Gamma(\Gamma^{|V|}(\emptyset)) = \Gamma^{|V|}(\emptyset)$$

A questo punto la nuova grammatica diventa:

- V' contiene i non-terminali selezionati nel modo del punto fisso:

$$V' \doteq \Gamma^{|V|}(\emptyset)$$

- P' contiene l'insieme di tutte le dimostrazioni di P, tale che il simbolo a sinistra della produzione appartenga a V' e α appartenga a $(V' \cup T)^*$:

$$P' \doteq \{A \to \alpha \in P \mid \underbrace{A \in V'}_{\subseteq P} \wedge \alpha \in (V' \cup T)^*\}$$

  In conclusione poiché P' considera tutte le produzioni di P per cui il simbolo A produce dei terminali, allora si ottiene che $\mathscr{L}(G) = \mathscr{L}(G')$.

▷ Cosa succede se l'operazione elimina il simbolo iniziale?
Significa che dal simbolo iniziale non si riesce a raggiungere sequenze di soli terminali, di conseguenza il linguaggio è vuoto. Dunque questa procedura dice se il linguaggio generato dalla grammatica è vuoto oppure no:

$$s \notin \Gamma^{|V|}(\emptyset) \iff \mathscr{L}(G) = \emptyset$$

Se la proprietà $\mathscr{L}(G) = \emptyset$ è vera, allora questa procedura è decidibile: se non si trova S alla fine della procedura (che termina un numero di volte, massimo pari alla cardinalità delle variabili nella grammatica), allora il linguaggio è sicuramente vuoto e viceversa.

Esempio

Si ha la seguente grammatica:

$$\begin{cases} S \to aSb \,|\, \varepsilon \,|\, H \\ H \to aH \end{cases}$$

La formula dell'operatore Γ consiste in:

$$\Gamma(W) = \{A \in V \mid \exists \alpha \in (T \cup W)^*. A \to \alpha \in P\}$$

Si parte da $W = \emptyset$ e si itera finché non è più possibile allargare l'insieme:

$$\Gamma(\emptyset) = \{A \in V \mid \exists \alpha \in (T \cup \emptyset)^*. A \to \alpha \in P\}$$

In questo caso guardando la grammatica, $\{S\}$ è l'unico simbolo ad avare terminali, pertanto $\Gamma(\emptyset) = \{S\}$. Si prosegue con l'iterazione:

$$\Gamma(\Gamma(\emptyset)) = \Gamma(\{S\}) = \{A \in V \mid \exists \alpha \in (T \cup \{S\})^*. A \to \alpha \in P\}$$

In S non ci sono altri simboli a possedere terminali, dunque $\Gamma(\Gamma(\emptyset)) = \Gamma(\{S\}) = \{S\}$ e si è arrivati al caso in cui un'iterazione ha come risultato l'iterazione precedente $\Gamma(\Gamma^{|V|}(\emptyset)) = \Gamma^{|V|}(\emptyset)$. In tal modo la nuova grammatica presenta solo i simboli trovati con queste iterazioni:

$$\begin{cases} S \to aSb \,|\, \varepsilon \,|\, \cancel{H} \\ \cancel{H \to aH} \end{cases} \quad = \quad S \to aSb \,|\, \varepsilon$$

Esempio

Si ha la seguente grammatica:

$$\begin{cases} S \to aSb \,|\, \varepsilon \,|\, H \\ H \to aH \\ N \to bN \,|\, b \\ R \to Na \end{cases}$$

La formula dell'operatore Γ consiste in:

$$\Gamma(W) = \{A \in V \mid \exists \alpha \in (T \cup W)^*. A \to \alpha \in P\}$$

Si parte da $W = \emptyset$ e si itera finché non è più possibile allargare l'insieme:

$$\Gamma(\emptyset) = \{A \in V \mid \exists \alpha \in (T \cup \emptyset)^*. A \to \alpha \in P\}$$

In questo caso guardando la grammatica, $\{S, N\}$ sono gli unici simboli ad avare terminali, pertanto $\Gamma(\emptyset) = \{S, N\}$. Si prosegue con l'iterazione:

$$\Gamma(\Gamma(\emptyset)) = \Gamma(\{S, N\}) = \{A \in V \mid \exists \alpha \in (T \cup \{S, N\})^*. A \to \alpha \in P\}$$

Il risultato corrisponde a simboli aventi sequenze di terminali oppure simboli contenenti $\{S, N\}$ non-terminali, perciò si ha che $\Gamma(\Gamma(\emptyset)) = \Gamma(\{S, N\}) = \{S, N, R\}$. Applicando infine un'ulteriore iterazione si arriva al punto fisso:

$$\Gamma(\Gamma(\Gamma(\emptyset))) = \Gamma(\{S, N, R\}) = \{A \in V \mid \exists \alpha \in (T \cup \{S, N, R\})^*. A \to \alpha \in P\}$$

Perché non ci sono più simboli aventi terminali e nemmeno contenenti ($\{S, N, R\}$ non-terminali: $\Gamma(\Gamma(\Gamma(\emptyset))) = \Gamma(\Gamma(\emptyset))$. Allora la nuova grammatica corrisponde a:

$$\begin{cases} S \to aSb \mid \varepsilon \mid \cancel{H} \\ \cancel{H \to aH} \\ N \to bN \mid b \\ R \to Na \end{cases} \quad = \quad \begin{cases} S \to aSb \mid \varepsilon \\ N \to bN \mid b \\ R \to Na \end{cases}$$

Esempio

Si ha la seguente grammatica:

$$\begin{cases} S \to H \\ H \to aH \mid B \\ B \to bB \mid A \\ A \to a \end{cases}$$

La formula dell'operatore Γ consiste in:

$$\Gamma(W) = \{A \in V \mid \exists \alpha \in (T \cup W)^*. A \to \alpha \in P\}$$

Si parte da $W = \emptyset$ e si itera finché non è più possibile allargare l'insieme:

$$\Gamma(\emptyset) = \{A \in V \mid \exists \alpha \in (T \cup \emptyset)^*. A \to \alpha \in P\}$$

In questo caso guardando la grammatica, $\{A\}$ è l'unico simbolo ad avare terminali, pertanto $\Gamma(\emptyset) = \{A\}$. Si prosegue con l'iterazione:

$$\Gamma(\Gamma(\emptyset)) = \Gamma(\{A\}) = \{A \in V \mid \exists \alpha \in (T \cup \{A\})^*. A \to \alpha \in P\}$$

Il risultato corrisponde a simboli aventi sequenze di terminali oppure simboli contenenti $\{A\}$ non-terminale, perciò si ha che $\Gamma(\Gamma(\emptyset)) = \Gamma(\{A\}) = \{A, B\}$. Applicando

un'ulteriore iterazione si arriva al punto fisso:

$$\Gamma(\Gamma(\Gamma(\emptyset))) = \Gamma(\{A, B\}) = \{A \in V \mid \exists \alpha \in (T \cup \{A, B\})^*. A \to \alpha \in P\}$$

Il risultato è uguale a simboli aventi sequenze terminali oppure simboli contenenti $\{A, B\}$ non-terminali, perciò si ha che $\Gamma(\Gamma(\Gamma(\emptyset))) = \Gamma(\{A, B\}) = \{A, B, H\}$. Infine tramite un'altra iterazione:

$$\Gamma(\Gamma(\Gamma(\Gamma(\emptyset)))) = \Gamma(\{A, B, H\}) = \{A \in V \mid \exists \alpha \in (T \cup \{A, B, H\})^*. A \to \alpha \in P\}$$

Il risultato è $\Gamma(\Gamma(\Gamma(\Gamma(\emptyset)))) = \Gamma(\{A, B, H\}) = \{A, B, H, S\}$. Dunque la nuova grammatica corrisponde alla grammatica iniziale; tutti i simboli sono utili.

Tramite questo lemma si trovano i simboli che generano solo terminali:

$$S \xrightarrow{G}{}^{*} \alpha x \beta \xrightarrow{G}{}^{*} \sigma \in T^*$$

Tuttavia non è detto che questa x sia raggiungibile da S e pertanto occorre vedere quali di questi siano raggiungibili da simboli iniziali.

Non solo si devono eliminare i simboli non-terminali non raggiungibili da S, ma eventualmente anche i simboli terminali che non vengono raggiunti da S.

Lemma 3.2.2: Eliminazione dei simboli non raggiungibili da S

Sia $G = \langle V, T, P, S\rangle$ una grammatica CF. Allora esiste $G' = \langle V', T', P', S\rangle$ tale che G' sia CF e per ogni x appartenente a terminali/non-terminali devono esistere due sequenze di simboli α, β appartenenti ai terminali/non-terminali tali che da S, con la grammatica G' e un certo numero di passi, si ottiene $\alpha x \beta$:

$$\forall x \in V' \cup T'. \exists \alpha, \beta \in (V' \cup T')^* : S \xrightarrow{G'}{}^{*} \alpha x \beta$$

Infine anche $\mathscr{L}(G) = \mathscr{L}(G')$.

Dimostrazione

Si costruisce un operatore Γ dalle parti di $V \cup T$ (insiemi di terminali/non-terminali) che va nelle parti di $V \cup T$, tale per cui $\Gamma(W)$ è definito come un simbolo x appartenente ai terminali/non-terminali oppure ε, tale che esiste un simbolo A non-terminale in $W \cap V$ così che da A in $\alpha x \beta$ appartenga a P:

$$\Gamma : 2^{V \cup T} \to 2^{V \cup T}, \quad \Gamma(W) = \{x \in V \cup T \cup \{\varepsilon\} \mid \exists A \in W \cap V \,.\, A \to \alpha x \beta \in P\} \cup \{S\}$$

Da questa costruzione si osserva che:

1. Se si applica $\Gamma(\emptyset)$, allora si ottiene l'insieme vuoto:

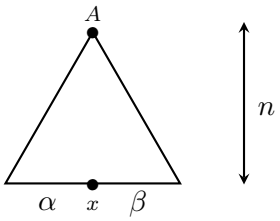
$$\Gamma(\emptyset) = \emptyset$$

Dunque si inizia a iterare da $W = \{S\}$.

2. Γ è monotono: se si allarga W, allora aumenta il numero di produzioni aventi la parte sinistra contenuta in W, quindi più x da considerare nell'insieme.

3. Il simbolo x appartiene all'iterazione n-esima a partire dall'insieme $\{S\}$ se e solo se da S esiste un albero di altezza n che ha $\alpha x \beta$ nella frontiera:

$$x \in \Gamma^n(\{S\}) \iff \exists \alpha, \beta \in (V \cup T)^*. S \xrightarrow{G}{}^* \alpha x \beta$$



Da queste note si deriva che Γ iterato alla $|V| + |T|$, a partire da $\{S\}$, è il punto fisso:

$$\Gamma^{|V|+|T|}(\{S\}) \to \textit{fix point}$$

Quindi si hanno come nuovi insiemi:

$$V' \doteq \Gamma^{|V|+|T|}(\{S\}) \cap V \subseteq V$$
$$T' \doteq \Gamma^{|V|+|T|}(\{S\}) \cap T \subseteq T$$
$$P' \doteq \{A \to \alpha \mid A \in V' \wedge \alpha \in (V' \cup T')^*\}$$

Ne consegue che $\mathscr{L}(G) = \mathscr{L}(G')$.

Esempio

Si considera la seguente grammatica:

$$\begin{cases} S \to aSb \,|\, \varepsilon \\ N \to bN \,|\, b \\ R \to Nc \end{cases}$$

Sapendo che $\Gamma(W)$ corrisponde a:

$$\Gamma(W) = \{x \in V \cup T \cup \{\varepsilon\} \mid \exists A \in W \cap V \,.\, A \to \alpha x \beta \in P\} \cup \{S\}$$

Si parte a iterare da $\{S\}$ vedendo tutti i simboli terminali/non-terminali raggiungibili:

$$\Gamma(\{S\}) = \{a, S, b, \varepsilon\}$$

Riapplicando Γ da questo insieme appena ottenuto, si ricava nuovamente il medesimo insieme:

$$\Gamma(\Gamma(\{S\})) = \{a, S, b, \varepsilon\} \rightarrow \textit{fix point}$$

In conclusione la grammatica diventa:

$$\begin{cases} S \rightarrow aSb \mid \varepsilon \\ \cancel{N \rightarrow bN \mid b} \\ \cancel{R \rightarrow Nc} \end{cases} \quad = \quad S \rightarrow aSb \mid \varepsilon$$

Esempio

Si considera la seguente grammatica:

$$\begin{cases} S \rightarrow aSb \mid \varepsilon \mid H \\ H \rightarrow cH \mid \varepsilon \\ D \rightarrow dD \mid \varepsilon \end{cases}$$

Sapendo che $\Gamma(W)$ corrisponde a:

$$\Gamma(W) = \{x \in V \cup T \cup \{\varepsilon\} \mid \exists A \in W \cap V \,.\, A \rightarrow \alpha x \beta \in P\} \cup \{S\}$$

Si parte a iterare da $\{S\}$ vedendo tutti i simboli terminali/non-terminali raggiungibili:

$$\Gamma(\{S\}) = \{S, a, b, \varepsilon, H\}$$

Riapplicando Γ da questo insieme appena ottenuto, si ricava un simbolo in più:

$$\Gamma(\Gamma(\{S\})) = \{S, a, b, \varepsilon, H, c\}$$

Infine se si riapplica l'iterazione, allora si ottiene il punto fisso:

$$\Gamma(\Gamma(\Gamma(\{S\}))) = \Gamma(\Gamma(\{S\})) = \{S, a, b, \varepsilon, H, c\}$$

In conclusione la grammatica diventa:

$$\begin{cases} S \rightarrow aSb \mid \varepsilon \mid H \\ H \rightarrow cH \mid \varepsilon \\ \cancel{D \rightarrow dD \mid \varepsilon} \end{cases} \quad = \quad \begin{cases} S \rightarrow aSb \mid \varepsilon \mid H \\ H \rightarrow cH \mid \varepsilon \end{cases}$$

Osservazione

Il primo lemma parte a iterare dalla frontiera (*bottom-up*), mentre il secondo parte a iterare dalla radice (*top-down*).

Teorema 3.2.2

Ogni linguaggio CF non vuoto è generato da una grammatica CF senza simboli inutili.

Dimostrazione

Supponendo di avere un linguaggio CF non vuoto, allora sia G una grammatica che genera questo linguaggio. Se da G si applica il 1° lemma, allora si ottiene G_1:

$$G \xrightarrow{\text{L1}} G_1$$

Successivamente se si applica il 2° lemma, allora si trova una grammatica G_2:

$$G_1 \xrightarrow{\text{L2}} G_2$$

Si suppone che in G_2 esistano simboli inutili $x \in V_2 \cup T_2$. Poiché questo simbolo x appartiene a $V_2 \cup T_2$, allora in virtù del 2° lemma esistono sequenze α, β di terminali/non-terminali tali che da S, nella grammatica G_2 e con un certo numero di passi, si vada in $\alpha x \beta$.

$$\xRightarrow{L2} \exists \alpha, \beta \in (V_2 \cup T_2)^*. S \xrightarrow{G_2}{}^* \alpha x \beta \ \wedge \ \mathscr{L}(G_1) = \mathscr{L}(G_2)$$

Inoltre si osserva che:

$$V_2 \subseteq V_1 \subseteq V, \quad T_2 \subseteq T_1 = T$$

Ciò implica che x appartenga a $V_1 \cup T_1$. A questo punto se x è un simbolo della grammatica G_1, allora significa che ha superato il filtraggio del 1° lemma.
Dunque per il 1° lemma esiste un σ appartenente a T_1^* tale che da $\alpha x \beta$, nella grammatica G_1 e con un certo numero di passi, si vada in σ:

$$\xRightarrow{L1} \exists \sigma \in T_1^*. \alpha x \beta \xrightarrow{G_1}{}^* \sigma \ \wedge \ \mathscr{L}(G) = \mathscr{L}(G_1)$$

Poiché $S \xrightarrow{G_2} \alpha x \beta$ e $\alpha x \beta \xrightarrow{G_1}{}^* \sigma$, se x è un simbolo raggiungibile da S e inoltre porta a simboli terminali, allora $\alpha x \beta$ va anche in σ con G_2:

$$S \xrightarrow{G_2} \alpha x \beta \xrightarrow{G_1}{}^* \sigma \Rightarrow \alpha x \beta \xrightarrow{G_2}{}^* \sigma$$

Effettuando una sostituzione si può affermare che:

$$S \xrightarrow{G_2} \alpha x \beta \xrightarrow{G_2}{}^* \sigma \in T_2^*$$

infine sapendo che $\mathscr{L}(G) = \mathscr{L}(G_1) = \mathscr{L}(G_2)$, allora implica che x non sia inutile.

3.3 Forma normale di Chomsky

La **forma normale di Chomsky** (FNC) è funzionale a costruire la dimostrazione del *pumping* lemma per i linguaggi CF. Dunque con l'intento di mostrare che ci sono linguaggi non CF.

3.3.1 Eliminazione delle ε-produzioni

Le ε-produzioni sono del tipo $A \to \varepsilon$ con $A \in V$ (variabili non-terminali). Prendendo come grammatica

$$G \begin{cases} S \to aSb \\ S \to \varepsilon \end{cases}$$

se $\varepsilon \in \mathscr{L}(G)$, allora è ammesso che $S \to \varepsilon \in P$. Altrimenti se si hanno delle produzioni che vanno in ε e coinvolgono un simbolo che non è quello iniziale, allora è opportuno eliminare quelle produzioni. L'eliminazione dell'ε-produzione consiste nell'eliminare tutti i simboli non terminali che generano la stringa vuota. Pertanto questi sono simboli che non contribuiscono alla costruzione di una stringa diversa dalla stringa vuota ε.

> Solo nel caso in cui la stringa vuota ε fosse nel linguaggio, allora si ammetterebbe una sola ε-produzione, che è quella del simbolo iniziale.

Si considera un simbolo $A \in V$ tale che da A, con una serie di iterazioni nella grammatica, si arriva a ε e A è diverso dal simbolo iniziale:

$$A \in V : A \xrightarrow{G}{}^{*} \varepsilon \wedge A \neq S$$

In questo modo la derivazione da A in ε è inutile, a meno che ε non sia interno al linguaggio.

$\triangleright$ Come vengono eliminate queste produzioni?
Se da A in $x_1 \dots x_n$ è una produzione della grammatica[4] da x_1, con un certo numero di passi in G, si arriva a ε e da x_n, con un certo numero di passi in G, si arriva a ε, allora anche da A, con un certo numero di passi in G, si arriva a ε:

$$A \to x_1, \dots x_n \in P \wedge \begin{array}{c} x_1 \xrightarrow{G}{}^{*} \varepsilon \\ \vdots \\ x_n \xrightarrow{G}{}^{*} \varepsilon \end{array} \Rightarrow A \xrightarrow{G}{}^{*} \varepsilon$$

Ciò significa che tutti i simboli $x_1 \dots x_n$ sono annullabili.

[4] La grammatica considerata è stata già filtrata da simboli inutili tramite i due lemmi.

Osservazione

Bisogna raccogliere tutti i simboli non-terminali che sono potenzialmente annullabili. Ovviamente $x_1 \dots x_n$ non sono terminali, altrimenti non potrebbero diventare ε.

Si costruisce una $\Gamma(W)$ corrispondente a una collezione di tutti i simboli A non-terminali, tale che esista una produzione avente il simbolo non-terminale a sinistra e se da α si sottraggono tutti i simboli di W allora si ottiene ε:

$$\Gamma(W) = \{A \in V \mid \exists A \to \alpha \in P \wedge \alpha \setminus W = \varepsilon\}$$

3.3.2 Eliminazione delle produzioni unitarie

Esistono ulteriori produzioni inutili da poter eliminare, come $A \to B$, chiamate **produzioni unitarie**. Si suppone di avere una grammatica di questo tipo:

$$\begin{cases} S \to aHb \\ S \to \varepsilon \\ H \to A \\ A \to C \\ A \to S \end{cases}$$

Queste ultime tre produzioni sono unitarie. Si potrebbero eliminare A e C facendo in modo che $H \to S$. Tuttavia l'H è presente anche in $S \to aHb$, perciò si può eliminare $H \to S$ e rimanere solo con la seguente grammatica:

$$\begin{cases} S \to aHb \\ S \to \varepsilon \\ \cancel{H \to A} \\ \cancel{A \to C} \\ \cancel{C \to S} \end{cases} \quad = \quad \begin{cases} S \to aSb \\ S \to \varepsilon \end{cases}$$

Questa procedura consiste nel ripercorrere all'indietro ciò che una produzione unitaria genera, passando dalla variabile di sinistra alla variabile di destra, in modo da ricondurle in produzioni non unitarie.

3.3.3 Concetto di "seguente"

Il "**seguente**" di una variabile A è l'insieme dei B tali che da A, nella grammatica G e con un certo numero di passi, si arriva in B e questo appartiene a V:

$$\text{seguente}(A) = \{B \mid A \xrightarrow{G}{}^{*} B \wedge B \in V\}$$

Nell'esempio in cui si ha la grammatica

$$\begin{cases} S \to aHb \\ S \to \varepsilon \\ H \to A \\ A \to C \\ A \to S \end{cases}$$

i **seguenti** di H sono A, C e S.

Poiché sono già stati eliminati i simboli inutili, allora non si hanno trasformazioni che riguardano V e T. Quindi si definisce una trasformazione che prende come grammatica $G = \langle V, T, P, S \rangle$ e si trasformano soltanto le produzioni P, cercando di accorpare le variabili in modo da ridurre il numero di produzioni: si eliminano le produzione unitarie. Si prende un certo P' da cui si sottraggono tutte le produzioni $A \to B$ che appartengono a P, con A e B variabili in V:

$$P' \doteq P \setminus \{A \to B \in P \mid A, B \in V\}$$

Successivamente per ogni B, appartenente ai seguenti in G del simbolo, tale che $B \to \beta$ sia una produzione della grammatica, allora P' corrisponde a P' unito $A \to B$:

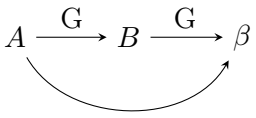
$$\forall B \in \text{seguenti}_G(A) \,.\, B \to \beta \in P : P' \doteq P' \cup \{A \to B\}$$

Esempio

Se si ha che:

$$A \xrightarrow{G} B \xrightarrow{G} \beta$$

Allora si collega A con β:



Questa trasformazione $A \to \beta$ effettua soltanto una chiusura transitiva lungo il cammino fatto da sole produzioni unitarie. Di conseguenza $\mathscr{L}(G) = \mathscr{L}(G')$, con G' che non possiede simboli inutili.

3.3.4 Forma normale di Chomsky

Teorema 3.3.3

Ogni linguaggio $\mathscr{L}$ CF che non contiene ε è generato da una grammatica con produzioni del tipo:

$$A \to BC, \quad \text{con } A, B, C \in V$$
$$A \to a, \quad \text{con } A \in V \text{ e } a \in T$$

Se una grammatica dispone di questa forma, allora si dice che la grammatica è in **forma normale di Chomsky**.

Dimostrazione

Si suppone una grammatica G CF senza:

- produzioni unitarie;
- ε-produzioni;
- simboli inutili (Lemma 1 e Lemma 2).

La trasformazione della grammatica avviene modificando le produzioni, cioè rimpiazzando con nuove produzioni ed eventualmente aggiungendo nuovi simboli.

▷ Come avviene questa trasformazione?
Se per ipotesi la grammatica G è CF con le varie riduzioni, allora la produzione generica ha la forma:

$$A \to x_1 \dots x_m \in P$$

- Se $m = 1$ e $x_1 \in T$, allora è in forma normale di Chomsky (FNC), perché è presente un solo simbolo ed è terminale.
- Se $m = 1$ e $x_1 \in V$, allora è impossibile, perché le produzioni unitarie sono state eliminate e questa sarebbe una produzione unitaria.
- Se $m = 2$ e $x_1, x_2 \in V$, allora è in forma normale di Chomsky (FNC), perché un simbolo non-terminale va in due non-terminali.
- Se $m \geq 2$ e $\exists i \in [1, m]$ tale che $x_i \in T$, allora per ogni $x_i \in T$:
  - Si aggiunge ai simboli non-terminali una nuova variabile B_i:

$$V \doteq V \cup \{B_i\}, \quad B_i = \texttt{new(V)}$$

`new`(V) genera una nuova variabile non-terminale rispetto a V.

- P diventa il vecchio insieme P unito a $B_i \to x_i$:

$$P \doteq P \cup \{\underbrace{B_i \to x_i}_{\text{FNC}}\}$$

- Si toglie da P la vecchia produzione $A \to x_1 \dots x_m$ e si unisce la nuova produzione $A \to y_1 \dots y_m$, dove y_i è uguale a x_i se $x_i \in V$, altrimenti è uguale a B_i se $x_i \in T$:

$$P \doteq (P \setminus \{A \to x_1 \dots x_m\}) \cup \{A \to y_1 \dots y_m\}$$

$$y_i = \begin{cases} x_i & x_i \in V \\ B_i & x_i \in T \end{cases}$$

Il risultato è che dopo aver iterato questa procedura, tutta la parte destra della produzione è formata da non-terminali: ogni terminale che si trova nella parte destra, si genera un nuovo non-terminale che rimpiazza il terminale e si aggiunge la produzione corrispondente.

- Se $m \geq 2$ e $\forall i \in [1, m]$ tale che $x_i \in V$, cioè si ha una produzione del tipo:

$$A \to \underbrace{x_1 \dots x_i \dots x_n}_{\in V}$$

Allora per andare in forma normale di Chomsky si accorpano i simboli. P diventa l'insieme vecchio P sottraendogli la vecchia produzione $A \to x_1x_2x_3 \dots x_m$ e unendo le nuove produzioni $A \to Bx_3 \dots x_m$ e $B \to x_1x_2$, dove B è una nuova variabile rispetto V:

$$P \doteq (P \setminus \{A \to x_1x_2x_3 \dots x_m\}) \cup \{A \to Bx_3 \dots x_m, B \to x_1x_2\},\ B = \texttt{new}(\text{V})$$

In questo modo la procedura può solo che terminare, perché l'algoritmo se trova una produzione con a destra sequenze di terminali/non-terminali, allora:

- aggiunge produzioni per accorpare i simboli;
- ottiene una produzione del tipo $A \to$ "sequenza di non-terminali".

Quando si ha una sequenza di non-terminali, vengono accorparti due a due e si itera questa procedura finché non si arriva alla forma normale di Chomsky.

Esempio

Si prende la seguente grammatica e da essa il linguaggio generato:

$$\mathscr{L}(G) = \{a^n c\, b^n \mid n \geq 0\}, \qquad G \begin{cases} S \to aSb \\ S \to c \end{cases}$$

- Si può notare che $\varepsilon \notin \mathscr{L}(G)$, poiché la stringa più piccola contiene solo c.
- Al momento $S \to c$ è FNC, mentre $S \to aSb$ non è FNC e di conseguenza la trasformazione deve trasformare $S \to aSb$.
- Si è nel caso in cui la produzione a destra possiede più di due simboli ($m \geq 2$) e due di questi sono terminali (a e b).

Allora per ogni simbolo terminale sulla parte destra della produzione:

- si genera un nuovo simbolo;
- si aggiunge la nuova produzione;
- si elimina la produzione originaria sostituendola con la nuova generata.

Dunque $S \to aSb$ genera:

$$\begin{cases} S \to \overset{\texttt{new}}{A}\ S\ \overset{\texttt{new}}{B} \\ A \to a \\ B \to b \\ S \to c \end{cases}$$

A questo punto le ultime tre produzioni sono in FNC, mentre la prima no. Di conseguenza in questo caso si applica nuovamente lo stesso procedimento, per cui $m \geq 2$ e i simboli terminali sono A, S in H:

$$\begin{cases} S \to HB \\ A \to a \\ B \to b \\ S \to c \\ H \to AS \end{cases}$$

Osservazione

La nuova grammatica ottenuta in esempio è equivalente a quella di partenza, ma possiede più simboli e produzioni consentendo di ricavare determinate proprietà esaminate nella Sezione 3.4.

3.4 Pumping lemma per linguaggi CF

Anche per i linguaggi CF esiste una proprietà sufficiente la quale afferma che se il linguaggio è CF, allora vale un certo "qualcosa". Quando si nega questo "qualcosa", allora si può dire che un certo linguaggio è CF.

Lemma 3.4.3: Pumping Lemma dei linguaggi CF

Sia $\mathscr{L}$ un linguaggio CF. Allora esiste $n \in \mathbb{N}$, dipendente da $\mathscr{L}$, tale che per ogni stringa $z \in \mathscr{L}$ tale per cui $|z| \geq n$ esistono $u, v, w, x, y \in T^*$ in modo che:

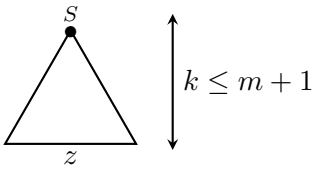
1. La stringa z è suddivisa in cinque sottostringhe: $z = uvwxy$.
2. Dentro vx c'è almeno un simbolo: $|vx| \geq 1$.
3. La dimensione di vwx è minore o uguale a n: $|vwx| \leq n$.
4. Per ogni $i \geq 0$ la stringa è pompata/spompata e appartiene a $\mathscr{L}$: $uv^iwx^iy \in \mathscr{L}$.

Dimostrazione

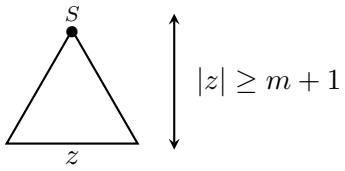
Sia $\mathscr{L}$ CF, allora implica che $\mathscr{L}$ sia uguale al linguaggio generato da $\mathscr{L}(G)$ e, poiché ogni grammatica può essere posta nella forma normale di Chomsky, si può assumere che $G = \langle V, T, P, S \rangle$ CF sia in FNC.
L'idea è quella di considerare m come la cardinalità dei simboli non-terminali:

$$m = |V| < \omega$$

$\triangleright$ Quanti alberi esistono che hanno come altezza $k \leq m + 1$?
Gli alberi nella forma di Chomsky sono tutti binari:

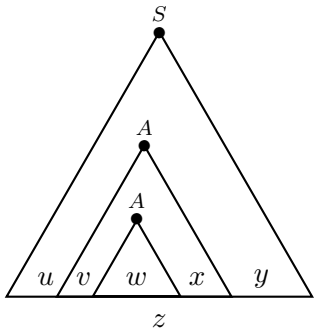


Esistono un numero finito di alberi, poiché fissata l'altezza $k \leq m + 1$ si sa che m è finito e di conseguenza pure il suo successore. A questo punto si prende come n la lunghezza della stringa massima ottenibile per ogni $k \leq m + 1$. Se l'albero è perfettamente bilanciato, allora $n = 2^{m+1}$ con $n \in \mathbb{N}$.
Dopodiché si prende una stringa $z \in \mathscr{L}$ tale che $|z| \geq n$, ossia una stringa nel linguaggio la cui lunghezza è più grande della lunghezza massima della stringa generabile con l'albero di altezza minore o uguale a $m + 1$. Quindi l'albero che genera la stringa z deve essere maggiore o uguale a $m + 1$:



Attenzione però che m è il numero di non-terminali e l'altezza di questo albero è $m+1$, ciò vuol dire che per generare z sicuramente è stato utilizzato più di una volta uno stesso non-terminale.

Sia A il penultimo albero che si ripete con al suo interno l'ultimo A. In questo modo si formano cinque sottoparti:

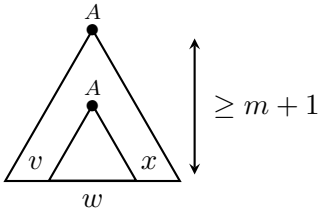


Le condizioni del *pumping* lemma sono tutte vere poiché:

1. C'è un non-terminale che si ripete più di una volta e di conseguenza si formano due sottoalberi con la stessa radice A, interni all'albero che genera la stringa z.

2. Per l'albero più grande con radice A esiste una produzione per cui $A \to BC$. Sia B e sia C andranno avanti: uno dei due arriverà nuovamente a produrre A, mentre l'altro proseguirà fino ad arrivare a un simbolo terminale. Quindi si è verificato che $|vx| \geq 1$, altrimenti verrebbero generate delle stringhe ε, che però non possono essere presenti in FNC.

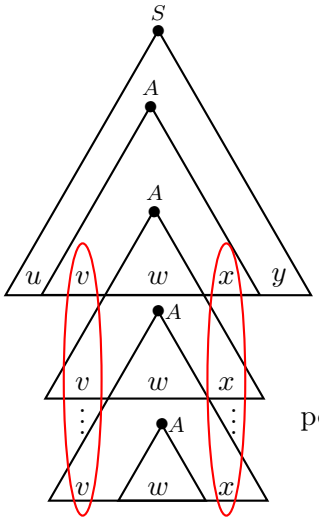
3. Se per assurdo $|vwx| > n$, allora l'albero di radice A, con dentro il suo sottoalbero di radice A, che genera vwx è più alto o uguale a $m+1$:



Ma allora questo albero non è quello più in "basso" che non contiene la ripetizione di altri simboli, poiché essendo di altezza maggiore o uguale a $m+1$

possiede simboli non-terminali che si ripetono. Ne consegue che non sarebbe l'albero minimo più vicino alla frontiera, ciò è assurdo e pertanto $|vwx| \leq n$.

4. Dato che $\mathscr{L}$ è CF, allora si può rimpiazzare l'albero A più piccolo con quello più grande e continuare a farlo per un certo numero di volte.



per i volte

In questa maniera v e x vengono pompate per i volte e la stringa uv^iwx^iy appartiene ancora a $\mathscr{L}$. Anche spompando si manterrebbe una stringa appartenente al linguaggio $\mathscr{L}$.

La memoria nelle grammatiche libere da contesto risiede nella struttura dei non-terminali. Infatti essendo i non-terminali un insieme finito, basta costruire una stringa abbastanza lunga da far sì che questa superi una funzione del numero di non-terminali e automaticamente ogni stringa più grande è decomponibile in cinque sottostringhe, tali che si riescano a pompare/spompare sempre due di queste.

Il *pumping* lemma dei linguaggi CF è utilizzato per dimostrare che un linguaggio non è CF:

$$\mathscr{L} \text{ è CF} \Rightarrow \text{PLCF} \iff \neg\text{PLCF} \Rightarrow \mathscr{L} \text{ non è CF}$$

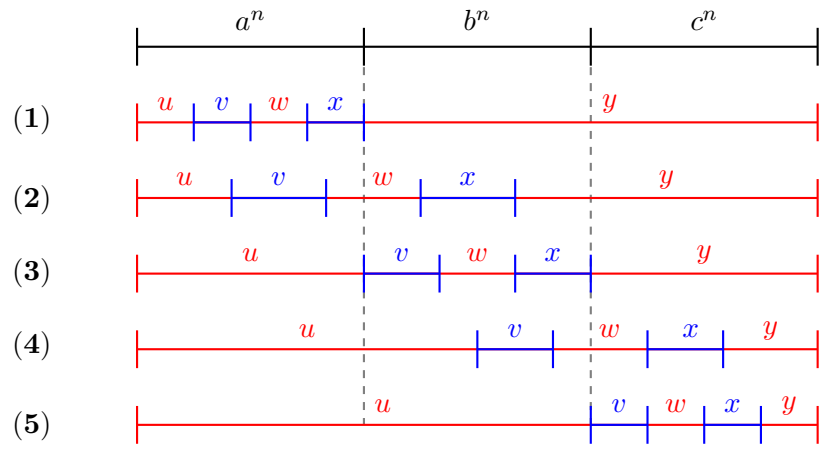
Esempio

Dato il linguaggio $\mathscr{L} = \{a^n b^n c^n \mid n \geq 1\}$, è un linguaggio CF?
Si prende una generica $n \in \mathbb{N}$ e la stringa $z = a^n b^n c^n$. Ciò implica che $|z| = 3n$ e $3n \geq n$. Per ogni suddivisione $u, v, w, x, y \in \{a, b\}^*$ tale per cui:

$$a^n b^n c^n = uvwxy, \quad |vx| \geq 1, \quad |vwx| \leq n$$

Tutto questo deve implicare che esista una $i \geq 0$ tale che $uv^iwx^iy \notin \mathscr{L}$.

▷ Quante possibili suddivisioni ci sono?
La stringa può essere suddivisa in cinque parti con i seguenti casi:



Si analizzano i casi individuati:

1. Pompando v e x si aumentano solo il numero di a e si rompe il linguaggio, perché la quantità di a supera quella di b e c.
2. La sottostringa vwx sta tra a^n e b^n. Quindi pompando le a e b la loro relazione con c si rompe.
3. Pompando v e x si aumenta solo il numero di b e si rompe il linguaggio, poiché la quantità di b è superiore rispetto a e c.
4. vwx sta tra b^n e c^n. Dunque pompando le b e c la loro relazione con a si rompe.
5. In conclusione per ogni suddivisione $i \geq 0$ tale che $uv^iwx^iy \notin \mathscr{L}$.

Questo implica che $\mathscr{L}$ non sia CF.

In realtà esistono altri 4 casi oltre quelli elencati nell'esempio appena svolto. Ovvero quando le sottostringhe x e v appartengono singolarmente a due sottoparti differenti:

- **Caso 1**: $v \in a^n$, un pezzo di $x \in a^n$ e un pezzo di $x \in b^n$.
- **Caso 2**: un pezzo di $v \in a^n$, un pezzo di $v \in b^n$ e $x \in b^n$.
- **Caso 3**: $v \in b^n$, un pezzo di $x \in b^n$ e un pezzo di $x \in c^n$.
- **Caso 4**: un pezzo di $v \in b^n$, un pezzo di $v \in c^n$ e $x \in c^n$.

In genere questi sono sottocasi dei "casi normali"[5] ispezionati nell'esempio e possono essere rotti con delle piccole accortezze: si sfruttano i vincoli imposti dal linguaggio oppure si riconducono ai corrispettivi "casi normali".

[5]Sono intesi quei casi in cui le sottostringhe x e v non appartengono singolarmente a due sottoparti differenti.

Osservazione

▷ È necessario dimostrare anche questi piccoli sottocasi?
Sì, perché la negazione del *pumping* lemma afferma che: per ogni suddivisione della stringa $z = uv^iwx^iy$, con $|z| \geq n$, esiste un pompaggio/spompaggio $i \in \mathbb{N}$ tale per cui $z \notin \mathscr{L}$.

▷ Quanti sottocasi possono esistere?
In linea generale a seconda di quante sottoparti sono presenti nella stringa z, si hanno:

$$\#\text{sottocasi} = (\#\text{sottoparti} - 1) \cdot 2$$

▷ Quanti "casi normali" possono esistere?
Anche in questo caso dipende dal linguaggio, ma in linea generale sono:

$$\#\text{casinormali} = (\#\text{sottoparti} \cdot 2) - 1$$

La stima dei casi totali può risultare:

$$\#\text{casitotali} = \#\text{casinormali} + \#\text{sottocasi} = \#\text{sottoparti} \cdot 4 - 3$$

Esempio

Dato il linguaggio $\mathscr{L} = \{a^{2^n} \mid n \geq 1\}$, è un linguaggio CF?
Si prende un generico $k \in \mathbb{N}$ e una stringa $z = a^{2^k}$ appartenente a $\mathscr{L}$. La lunghezza della stringa corrisponde a $|z| = 2^k \geq k$.
Allora si prende $z = uvwxy$ tale che $v = a^p$, $x = a^q$ e $p + q \geq 1$. Ciò deriva dalla condizione per cui $|vx| \geq 1$.

▷ Cosa succede se si pompa v e x?
La lunghezza di uv^nwx^ny è uguale alla lunghezza di $uvwxy$ sommata alle lunghezze di v e x per $n - 1$ volte:

$$|uv^nwx^ny| = \underbrace{uvwxy}_{z} + (n-1)|v| + (n-1)|x| = 2^k + (n-1) \cdot (|v| + |x|)$$

È noto che le lunghezze di v e x sono rispettivamente p e q:

$$2^k + (n-1) \cdot (|v| + |x|) = 2^k + (n-1) \cdot (p+q)$$

Perciò bisogna scegliere un $n \in \mathbb{N}$ per cui l'ultima espressione ottenuta non sia

potenza di 2. Allora si sceglie $n = 2^{k+1} + 1$ così da avere:

$$\begin{aligned} 2^k \cdot (2^{k+1} + 1 - 1) \cdot (p + q) &= 2^k + 2^{k+1}(p + q) \\ &= 2^k + 2^k \cdot 2 \cdot (p + q) \\ &= 2^k \cdot (2(p + q) + 1) \end{aligned}$$

Da questa espressione si nota che $p + q \geq 1 \in \mathbb{N}$ per ipotesi e un numero naturale moltiplicato per 2 e sommato a 1 è sempre un numero dispari. Di conseguenza $2^k(2(p+q)+1)$ non può essere una potenza di 2: $uv^{2^{k+1}+1}wx^{2^{k+1}+1}y \notin \mathscr{L}$ e $\mathscr{L}$ non è un linguaggio CF.

3.4.1 Proprietà di chiusura dei linguaggi CF

Per i linguaggi CF le proprietà di chiusura sono differenti rispetto i linguaggi regolari.

Teorema 3.4.4

I linguaggi CF sono chiusi per unione, concatenazione e iterazione (* di Kleene).

Dimostrazione

Bisogna costruire le singole grammatiche per tutte le proprietà. Innanzitutto si ricorda che la cardinalità dei linguaggi CF è infinita: $|\text{CF}| = \omega$; ed è numerabile: per ogni linguaggio CF si ha una grammatica formata da una sequenza finita di simboli. Essendo numerabili si effettua l'induzione.
Siano $\mathscr{L}_1 = \mathscr{L}(G_1)$ e $\mathscr{L}_2 = \mathscr{L}(G_2)$, con le grammatiche:

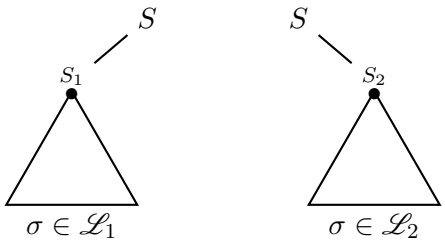
$$\begin{aligned} G_1 &= \langle V_1, T_1, P_1, S_1 \rangle \\ G_2 &= \langle V_2, T_2, P_2, S_2 \rangle \end{aligned}$$

Inoltre si suppone che gli insiemi tra G_1 e G_2 siano disgiunti, ovvero che ciascuna grammatica contenga nomi di variabili diverse.

- La **grammatica unione** tra le due grammatiche corrisponde a:

$$G_\cup = \langle V_1 \cup V_2 \cup \underbrace{\{S\}}_{\texttt{new}}, T_1 \cup T_2, P_1 \cup P_2 \cup \{S \to S_1 \mid S_2\}, S \rangle$$

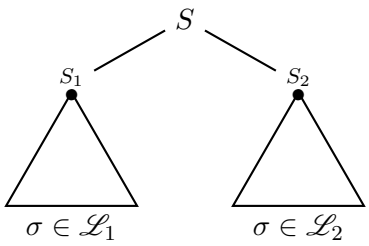
  dove nell'insieme delle variabili non-terminali viene aggiunto un nuovo simbolo $\{S\}$, mentre nell'insieme delle produzioni viene aggiunta una produzione tale che sia l'unione dei due linguaggi: si può andare in S_1, che genera l'albero per i linguaggi $\mathscr{L}_1$, oppure in S_2, che genera l'albero per i linguaggi $\mathscr{L}$.



- La **grammatica concatenazione** tra le due grammatiche corrisponde a:

$$G_\bullet = \langle V_1 \cup V_2 \cup \underbrace{\{S\}}_{\texttt{new}}, T_1 \cup T_2, P_1 \cup P_2 \cup \{S \to S_1 S_2\}, S \rangle$$

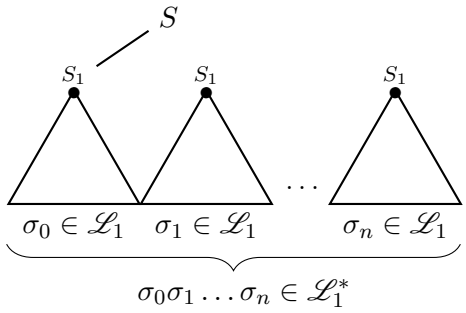
  dove nell'insieme delle variabili non-terminali viene aggiunto un nuovo simbolo $\{S\}$, mentre nell'insieme delle produzioni viene aggiunta una produzione tale che sia la concatenazione dei due linguaggi.



- La **grammatica iterazione** è composta come:

$$G_* = \langle V_1 \cup \underbrace{\{S\}}_{\texttt{new}}, T_1, P_1 \cup \{S \to \varepsilon \,|\, S_1 S\}, S \rangle$$

  dove nell'insieme delle variabili non-terminali viene aggiunto un nuovo simbolo $\{S\}$, mentre nell'insieme delle produzioni viene aggiunta una produzione tale che consenta di ripetere l'albero S_1 oppure di fermarsi con ε.



I linguaggi CF non sono chiusi per intersezione. Per verificarlo basta trovare come con-

troesempio (di questa proprietà) due linguaggi che se intersecati allora non restituiscono un linguaggio CF. Per esempio $a^n b^n c^n$ non è un linguaggio CF e se prendendo l'intersezione di due linguaggi CF si ottiene $a^n b^n c^n$, allora la proprietà di intersezione non è valida.

Dimostrazione

Sia $\mathscr{L}_1 = \{a^n b^n c^j \mid n, j \geq 1\}$, dove il numero di a e b è vincolato a rimanere uguale, mentre il numero di c è svincolato dalla quantità n. Dunque si costruisce la grammatica per questo linguaggio:

$$G_1 \begin{cases} S \to RC \\ R \to ab \,|\, aRb \\ C \to c \,|\, cC \end{cases}$$

Successivamente sia $\mathscr{L}_2 = \{a^j b^n c^n \mid j, n \geq 1\}$, dove la situazione è analoga a $\mathscr{L}_1$ con la sola differenza che il vincolo è tra le b e c, mentre a è svincolata. Anche in questo casi si costruisce la grammatica:

$$G_2 \begin{cases} S \to AR \\ A \to a \,|\, aA \\ R \to bc \,|\, bRC \end{cases}$$

Allora l'intersezione significa che esistono stringhe stanno sia in $\mathscr{L}_1$ sia in $\mathscr{L}_2$:

$$\mathscr{L}_1 \cap \mathscr{L}_2 = \{a^n b^n c^j \mid j, n \geq 1\} \cap \{a^j b^n c^n \mid j, n \geq 1\}$$

Tuttavia le stringhe comuni a entrambi i linguaggi sono quando $j = n$, perché b^n è in comune per ciascun linguaggio:

$$\mathscr{L}_1 \cap \mathscr{L}_2 = \{a^n b^n c^n \mid n \geq 1\}$$

Ecco che però $a^n b^n c^n$ non è CF e di conseguenza l'intersezione di due linguaggi CF non restituisce sempre un altro linguaggio CF.

Poiché l'intersezione non è valida, ne consegue che nemmeno la complementazione lo sia. Altrimenti per De Morgan varrebbe che $\mathscr{L}_1 \cap \mathscr{L}_2 = \overline{\overline{\mathscr{L}}_1 \cup \overline{\mathscr{L}}_2}$.

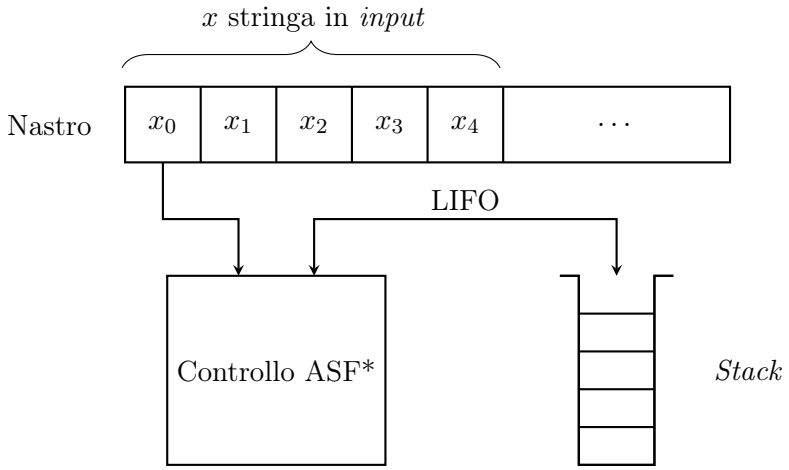
In linea generale si ha questa situazione nelle proprietà di chiusura:

Operazione	REG	CF
$\cup$-finita	✓	✓
$\cap$-finita	✓	??
Complemento	✓	??
$\cup$-infinita	??	??
$\cap$-infinita	??	??
Iterazione	✓	✓
Concatenazione	✓	✓

Dove "??" indica che non è sempre certo il risultato dell'operazione.

3.5 Automi a pila

Un **automa a pila** è formato da un **controllo ASF***[6] che può soltanto leggere i simboli dal nastro ed effettuare operazioni di lettura/scrittura in una memoria chiamata **pila** (*stack*). Questa memoria non ha limite in altezza, di conseguenza non può andare in *buffer overflow*, e adotta una politica di lettura/scrittura LIFO (*Last-In First-Out*).



Le possibili azioni della macchina sono:

- Leggere dal nastro in sola lettura un simbolo e la testa dello *stack* (`pop`).
- Fare una transizione di stato del controllo.
- Scrivere sulla pila una sequenza (`push`).

Matematicamente un automa a pila è definito come:

$$M = \langle Q, \Sigma, R, t, q_0, F \rangle$$

[6] Il simbolo * indica che può essere un automa D, ND oppure ND-ε.

- Q è un insieme finito di stati.
- Σ è l'alfabeto dei simboli del nastro.
- R è l'alfabeto dei simboli dello *stack*.
- t è la relazione di transizione.
- $q_0 \in Q$ è lo stato iniziale nell'automa.
- $z_0 \in R$ è il simbolo iniziale nello *stack*.
- $F \subseteq Q$ è l'insieme finito di stati finali.

Nel dettaglio viene presentato un ASFND-ε con la pila, pertanto la relazione t è definita come: dallo stato Q si legge un simbolo da nastro oppure stringa vuota (ε) e poi uno simbolo dallo *stack*, dopodiché si genera un insieme finito che va dalle parti di Q per R^*:

$$t : Q \times (\Sigma \cup \{\varepsilon\}) \times R \to 2^{Q \times R^*}$$

Quindi si suppone di essere nello stato iniziale $q_0 \in Q$ e di leggere un simbolo $x_0 \in \Sigma$ e $z_0 \in R$:

- la testina si sposta sul successivo simbolo del nastro x_1;
- lo stato del controllo cambia in q_1;
- la pila viene caricata con uno o una sequenza di simboli.

> Ogni volta che si è in uno stato e vengono letti due simboli, rispettivamente uno per Σ e uno per R, allora si genera un nuovo stato in Q e una sequenza finita deterministica in R^*.

3.5.1 Descrizione istantanea della macchina

Per **descrizione istantanea** di M si intende: la struttura[7] della sequenza di simboli in lettura sul nastro, lo stato corrente e il contenuto della pila. Dunque questa descrizione può essere intesa come l'attuale situazione della macchina ed è definita nella tripla (q, x, γ):

- q è lo stato corrente;
- $x \in \Sigma^*$ è la stringa in *output* ancora da leggere;
- $\gamma \in \mathbb{R}$ è il contenuto dello *stack*.

[7] I simboli rimanenti da dover ancora leggere.

Una singola computazione dell'automa consiste in:

$$(q, x, \gamma) = (q, aw, z\alpha) \xrightarrow{M}{}^{1} (p, w, \beta\alpha), \quad \text{se } (p, \beta) \in t(q, a, z)$$

Se si legge il simbolo a, si è nello stato q e in testa allo *stack* c'è il solo simbolo z, allora la transizione cambia lo stato da q a p e inserisce nello *stack* la sequenza β al posto di z.

Pertanto una computazione di un automa a pila è una sequenza finita, perché per ogni *input* vengono consumati tutti i simboli sul nastro, arrivando a due possibili situazioni:

1. Il linguaggio accettato per **stati finali** di un automa a pila M è l'insieme di tutte le stringhe x appartenenti a Σ^*, tali che partendo da q_0 e leggendo x sul nastro e z in cima allo *stack*, con un sequenza finita di transizioni, si ottiene uno stato $q \in F$ con una stringa rimanente vuota (ε) e $\gamma \in R^*$ sulla pila:

$$\mathscr{L}_F(M) = \{x \in \Sigma^* \mid (q_0, x, z_0) \xrightarrow{M}{}^{*} (q, \varepsilon, \gamma),\ \gamma \in R^*,\ q \in F\}$$

2. Il linguaggio accettato per **pila vuota** è l'insieme di tutte le stringhe x in Σ^*, tale che partendo da q_0 e dovendo leggere x sul nastro e z_0 sulla pila, con un numero finito di passi dell'esecuzione dell'automa, si ottiene un qualsiasi stato $q \in Q$ con una stringa rimanente vuota (ε) e lo *stack* vuoto:

$$\mathscr{L}_P(M) = \{x \in \Sigma^* \mid (q_0, x, z_0) \xrightarrow{M}{}^{*} (q, \varepsilon, \varepsilon),\ q \in Q\}$$

▷ I linguaggi accettati per $\mathscr{L}_F$ e $\mathscr{L}_P$ coincidono?
Sì, i due metodi per accettare un linguaggio con un ε-APND sono equivalenti.

Dimostrazione

Si mostra l'uguaglianza da entrambi i lati:

- Sia M un ε-APND. Allora esiste un M' ε-APND tale che il linguaggio accettato per pila vuota di M' sia uguale al linguaggio accettato per stato finale di M:

$$\Rightarrow \exists M'(\varepsilon\text{-APND}) : \mathscr{L}_P(M') = \mathscr{L}_F(M)$$

  Per fare ciò basta effettuare, una volta esaurita la stringa del nastro, delle ε-transizioni per svuotare lo *stack*:

$$(q_0, x, z_0) \underbrace{\xrightarrow{M}{}^{*}}_{=M'} (q, \varepsilon, \gamma) \underbrace{\xrightarrow{M'}{}^{*}}_{\varepsilon\text{-transizioni}} (q', \varepsilon, \varepsilon)$$

- Sia M un ε-APND. Allora esiste un M' ε-APND tale che il linguaggio accettato per pila vuota di M sia uguale al linguaggio finale di M':

$$\Rightarrow \exists M'(\varepsilon\text{-APND}) : \mathscr{L}_P(M) = \mathscr{L}_F(M')$$

Per fare ciò occorre aggiungere uno stato finale q_f in M' ed effettuare un ulteriore passaggio così da andare in q_f:

$$(q_0, x, z_0) \underbrace{\xrightarrow{M}{}^{*}}_{=M'} (q, \varepsilon, \gamma) \underbrace{\xrightarrow{M'}{}^{1}}_{\varepsilon\text{-transizioni}} (q_f, \varepsilon, \varepsilon)$$

Esempio

Viene dato il linguaggio $\mathscr{L} = \{wcw^R \mid w \in \{\mathrm{a}, \mathrm{b}\}^*\}$ e l'alfabeto $\Sigma = \{\mathrm{a}, \mathrm{b}, \mathrm{c}\}$. Questo linguaggio genera stringhe palindrome con un carattere centrale 'c', il quale non può apparire in w.

▷ Qual è l'automa a pila deterministico in grado di riconoscere $\mathscr{L}$?

- L'alfabeto dello *stack* è $R = \{z, A, B\}$, nel quale z corrisponde al simbolo iniziale, mentre A, B servono per memorizzare nello *stack* i simboli in lettura dal nastro, rispettivamente 'a' e 'b'.
- Gli stati presenti in Q sono: q_0 che carica la pila e q_1 che scarica la pila.
- L'insieme degli stati finali F è vuoto, perché si vuole accettare il linguaggio per pila vuota.
- q_0 è lo stato iniziale.
- z è il simbolo iniziale dello *stack*.

Si stabilisce una tabella di transizioni dell'automa:

q_0	ε	'a'	'b'	'c'
z		q_0, zA	q_0, zB	q_1, ε
A				
B				

q_1	ε	'a'	'b'	'c'
z				q_1, ε
A		q_1, ε		
B			q_1, ε	

Un possibile scenario che si può costruire è:

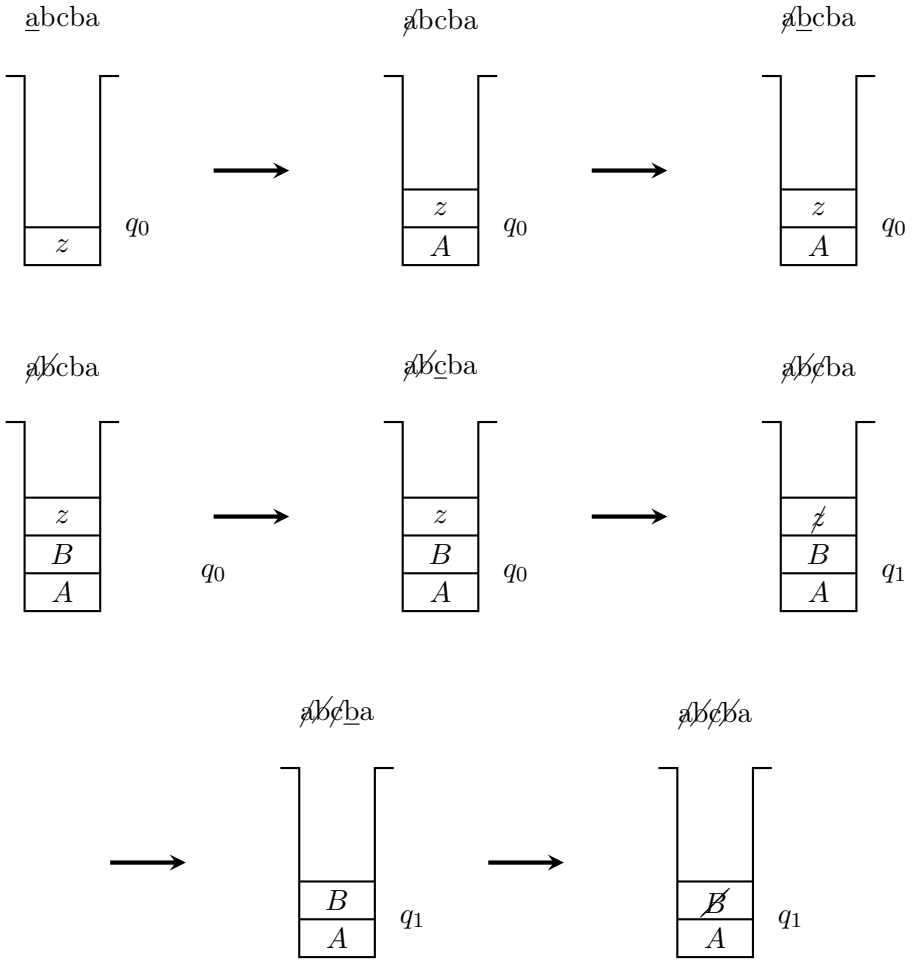
1. La stringa inizia con 'a' o 'b'. Se si trova una 'a' in lettura e lo *stack* ha il simbolo iniziale z, allora si rimane in q_0 e si scarica il simbolo z per poi inserire A con sopra z.
   - Se si legge 'b', allora l'automa lavora in egual modo, ma inserendo B al posto di A.
2. Se si legge il simbolo 'c', allora significa che la stringa w è stata consumata. Di conseguenza i simboli sottostanti a z nello *stack* compongono la stringa w invertita, ossia w^R.

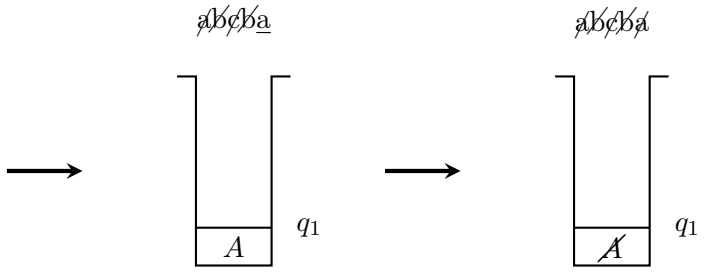
- Quindi si consuma z all'interno dello *stack* e si effettua una transizione verso lo stato q_1.

3. A questo punto in q_1 se si legge 'a' e in testa alla pila è presente A, allora si scarica quel simbolo rimanendo in q_1.

   - Analogamente vale per 'b' scaricando il simbolo B nello *stack*.

La stringa appartiene al linguaggio se quando si arriva alla conclusione lo *stack* risulta essere vuoto, altrimenti si presentano situazioni per cui l'automa non può proseguire (celle vuote della tabella di transizioni).





Esempio

Viene dato il linguaggio $\mathscr{L} = \{ww^R \mid w \in \{\mathrm{a},\mathrm{b}\}^*\}$ e l'alfabeto $\Sigma = \{\mathrm{a},\mathrm{b}\}$. In questo caso si ha una stringa palindroma senza alcun separatore intermedio come nell'esempio precedente.

▷ Qual è l'automa a pila in grado di riconoscere $\mathscr{L}$?
Osservando la stringa si sa per forza che in mezzo si trovano due caratteri 'a' o 'b':

$$\ldots \mathrm{aa} \ldots \quad \text{oppure} \quad \ldots \mathrm{bb} \ldots$$

Tuttavia l'automa, se deterministico, non può sapere con certezza quando una coppia di caratteri 'a' oppure 'b' è nel centro della stringa in *input*. Di conseguenza occorre utilizzare un automa non deterministico, così quando si incontra una coppia si prosegue in due ramificazioni: una considera il caso che la coppia sia centrale, mentre l'altra prosegue scartando la centralità. Se una delle varie ramificazioni svuota lo *stack* avendo concluso la stringa in *input*, allora essa appartiene al linguaggio.

- L'insieme Q è composto da: q_0 che carica la pila e q_1 che scarica la pila.
- L'insieme R dei simboli dello *stack* contiene z (simbolo iniziale), A e B (rispettivamente per 'a' e 'b').
- $z \in R$ è il simbolo iniziale dello *stack*.

Si stabilisce una tabella di transizioni dell'automa:

q_0	ε	'a'	'b'
z		q_0, Az	q_0, Bz
A		q_0, AA q_1, ε	q_0, BA
B		q_0, AB	q_0, BB q_1, ε

q_1	ε	'a'	'b'
z	q_1, ε		
A		q_1, ε	
B			q_1, ε

Un possibile scenario che si può costruire è:

1. Se si legge 'a' dal nastro e z dallo *stack*, allora si rimane in q_0 e si carica z con sopra A nello *stack*.

   - Se si legge 'b' dal nastro e z dallo *stack*, allora si rimane in q_0 e si carica z con sopra B nello *stack*.

   In questa computazione z sta sempre in fondo, così nella successiva lettura può verificarsi una doppia di A oppure B.

2. Se in cima allo *stack* si legge A e dal nastro 'a', allora questa può essere potenzialmente una coppia centrale nella stringa e avviene una diramazione non deterministica:

   - Si consuma A rimanendo in q_0 e nello *stack* si carica A con sopra A.
   - Si scarica A andando nello stato q_1.

2.1. Diversamente se si legge A dallo *stack*, ma 'b' dal nastro, allora si consuma A e si carica nello *stack* A con sopra B.

3. Se in cima allo *stack* si legge B e dal nastro 'b', allora avviene una diramazione non deterministica come nel punto (2).
   Diversamente se dal nastro si legge 'a', allora si perde la possibile coppia: si consuma B e si carica nello *stack* A con sopra B.

4. Per lo stato q_1 esistono solo due casi possibili per scaricare i simboli della pila:

   - Leggere A dallo *stack* e 'a' dal nastro.
   - Leggere B dallo *stack* e 'b' dal nastro.

5. Infine se viene letto il simbolo z dallo *stack*, allora la stringa è per forza conclusa e si effettua una ε-transizione che scarica il simbolo z dallo *stack*.

Osservazione

▷ Che differenza c'è tra $\mathscr{L}_1 = \{wcw^R \mid w \in \{\mathrm{a},\mathrm{b}\}^*\}$ e $\mathscr{L}_2 = \{ww^R \mid w \in \{\mathrm{a},\mathrm{b}\}^*\}$?

L'unica differenza è il simbolo 'c' presente in $\mathscr{L}_1$ e non in $\mathscr{L}_2$, il quale implica il determinismo per $\mathscr{L}_1$ e il non-determinismo per $\mathscr{L}_2$. Tuttavia nella realtà l'automa deterministico è realizzabile, mentre quello non-deterministico avrebbe una dimensione non lineare, bensì esponenziale, pertanto irrealizzabile.
Dunque $\mathscr{L}(\text{APD}) \subseteq \mathscr{L}(\varepsilon\text{-APND})$, ma non possono riconoscere gli stessi linguaggi:

$$\mathscr{L}(\text{APD}) \neq \mathscr{L}(\varepsilon\text{-APND})$$

$\mathscr{L}$(APD) sono i ***parser***[8], mentre $\mathscr{L}$(ε-APND) sono i CF.

3.5.2 Automa a pila corrispondente a una grammatica CF

Lemma 3.5.4: Linguaggio accettato da un ε-APND a singolo stato

Se $\mathscr{L}$ è il linguaggio accettato per pila vuota da un M ε-APND, allora $\mathscr{L}$ corrisponde al linguaggio accettato per pila vuota da un M' ε-APND avente un solo stato.

$\mathscr{L} = \mathscr{L}_P(M), \quad M = \varepsilon\text{-APND} \Rightarrow \mathscr{L} = \mathscr{L}_P(M'),\ M' = \varepsilon\text{-APND, con 1 stato finale}$

Questo lemma mostra che l'automa, il quale gestisce il controllo della macchina, è ininfluente: una volta che si possiede una pila arbitrariamente grande, allora la transizione di stato è inutile. Quindi la transizione di stato può essere simulata tramite una particolare gestione della memoria: si virtualizza il controllo dell'automa a pila dentro la gestione della memoria.

Dimostrazione

Si codificano gli stati Q con i simboli della pila. Quindi l'automa M consiste in:

$$M = \langle Q, \Sigma, R, q_0, z_0, t, \emptyset \rangle$$

Mentre M' corrisponde a:

$$M' = \langle \{q\}, \Sigma, R', q, z'_0, t', \emptyset \rangle$$

La virtualizzazione è "giocata" sull'espandere i simboli di pila in R'. Allora si pone R' come il prodotto cartesiano tra R e Q:

$$R' = R \times Q$$

Un oggetto dentro R' è una coppia $(q_i \in Q, z_i \in R) \in R'$, dove $z'_i = (q_i, z_i)$.

$\triangleright$ Come funzionano le computazioni in M e M'?

- **Computazioni in** M: si ha una descrizione istantanea:
  - parte dallo stato q_{ij};
  - si deve leggere la stringa aw, dove a è il primo simbolo in lettura;
  - in testa alla pila è presente z_i, seguita da una stringa α composta da simboli di pila.

[8] Fase di analisi sintattica del compilatore.

$$(q_i, aw, z_i\alpha)$$

M esegue una computazione andando in:

- un nuovo stato q_j;
- rimane solo la stringa w, perché il simbolo a è stato consumato;
- sulla pila è stata caricata una sequenza di simboli γ sopra α.

$$(q_i, aw, z_i\alpha) \xrightarrow{M}{}^{1} (q_j, w, \underbrace{z_1 \dots z_n}_{\gamma}\alpha)$$

- **Computazione in M'**: la computazione in M' avviene solo se si ha la transizione:

$$(q, aw, \underbrace{(q_i, z_i)}_{z_i'}\alpha) \xrightarrow{M'} (q, w, \gamma'\alpha)$$

dove z_i' è la coppia che appartiene a R', mentre la nuova sequenza γ' consiste nelle coppie:

$$\gamma' = (q_j, z_1)(q_j, z_2)\dots(q_j, z_n)$$

Il fattore più importante è quello che nella computazione M' lo stato q rimane invariato.

Questo metodo riduce la transizione di stato a una verifica sullo *stack*: quando si effettua `pop` sulla testa della pila, allora si ottiene l'informazione dello stato di M.

Teorema 3.5.5

Se $\mathscr{L} = \mathscr{L}_P(M)$ con M un ε-APND avente un solo stato. Allora $\mathscr{L}$ è CF.

Dimostrazione

Si suppone che M sia un ε-APND avente un solo stato:

$$M = \langle \{q\}, \Sigma, R, t, q, z_0, \emptyset \rangle$$

Da questo automa si ricava una grammatica CF:

$$G = \langle R, \Sigma, z_0, P \rangle$$

- I simboli di pila R sono presi come simboli non-terminali.
- L'alfabeto Σ della macchina corrisponde all'insieme di simboli terminali.

- Il simbolo iniziale è posto uguale al simbolo iniziale di pila.
- Le produzioni P sono composte come:

$$\underbrace{z \to az_1 \dots z_n \in P}_{\text{Produzione CF}} \iff (q, z_1 \dots z_n) \in t(q, a, x)$$

  Quindi si carica nella pila $z_1 \dots z_n$ avendo letto a dal nastro e trovando z in testa alla pila. Tutto ciò corrisponde alla produzione che trasforma il simbolo in testa alla pila in un simbolo terminale a seguito dalla sequenza di simboli non-terminali $z_1 \dots z_n$.

Per esempio si pone di avere in lettura sul nastro il simbolo a e in pila $z\alpha$. Con un passo in M si ottiene la consumazione di a e il caricamento di $z_1 \dots z_n\alpha$ in pila:

$$(q, a, z\alpha) \xrightarrow{M}{}^{1} (q, \varepsilon, z_1 \dots z_n\alpha)$$

Ciò avviene se e solo se da $z\alpha$, con una grammatica G, si va a $az_1 \dots z_n\alpha$:

$$(q, a, z\alpha) \xrightarrow{M}{}^{1} (q, \varepsilon, z_1 \dots z_n\alpha) \iff z\alpha \xrightarrow{G}{}^{1} az_1 \dots z_n\alpha$$

Per induzione si nota che, per tutte le stringhe x appartenenti a Σ^*, si ha $(q, x, z_0) \xrightarrow{M}{}^{*} (q, \varepsilon, \varepsilon)$ se e solo se da z_0, nella grammatica G, si arriva a soli terminali x:

$$\forall x \in \Sigma^* : (q, x, z_0) \xrightarrow{M}{}^{*} (q, \varepsilon, \varepsilon) \iff z_0 \xrightarrow{G}{}^{*} x$$

Pertanto nell'automa viene letta una sequenza di simboli e nella grammatica viene prodotta.

3.6 Forma normale di Greibach

Definizione 3.6.1: Forma normale di S.Greibach (1965)

Una grammatica CF è in forma normale di Greibach (FNG) se le produzioni sono della forma $A \to az_1 \dots z_n$, dove $a \in T$ e $z_1 \dots z_n \in V$.

Esempio

Si suppone che G sia FNG e sia formata come:

$$\begin{array}{lll} S \to \mathrm{a}B & A \to \mathrm{a}S & B \to \mathrm{b}S \\ S \to \mathrm{b}A & A \to \mathrm{b}AA & B \to \mathrm{b} \\ & A \to \mathrm{a} & B \to \mathrm{a}BB \end{array}$$

Si osserva che l'automa non è deterministico, poiché per A e B ci sono almeno due produzioni aventi lo stesso simbolo terminale.

$\triangleright$ Quali sono le transizioni per l'automa con la stringa "aababb"?
Si parte dallo stato q (unico) e nello *stack* è presente soltanto il simbolo iniziale S:

1. Si legge dal nastro il simbolo 'a' e si consuma dallo *stack* il simbolo S per poi caricare B:

   Produzione: $S \to \mathrm{a}B$
   Transizione: $(q, \cancel{\mathrm{a}}\mathrm{ababb}, \cancel{S}) = (q, \mathrm{ababb}, B)$

2. Si legge dal nastro il simbolo 'a' e si consuma dallo *stack* il simbolo B per poi caricare BB:

   Produzione: $B \to \mathrm{a}BB$
   Transizione: $(q, \cancel{\mathrm{a}}\mathrm{babb}, \cancel{B}) = (q, \mathrm{babb}, BB)$

3. A questo punto dal nastro si legge 'b' e sono possibili due scelte (non determinismo):

   Produzione (1): $B \to \mathrm{b}$
   Transizione (1): $(q, \cancel{\mathrm{b}}\mathrm{abb}, \cancel{B}B) = (q, \mathrm{abb}, B)$
   Produzione (2): $B \to \mathrm{b}S$
   Transizione (2): $(q, \cancel{\mathrm{b}}\mathrm{abb}, \cancel{B}B) = (q, \mathrm{abb}, SB)$

4. Si prosegue per entrambe le ramificazioni leggendo lo stesso simbolo 'a' dal nastro e un simbolo diverso sullo *stack*:

   Produzione (1): $B \to \mathrm{a}BB$
   Transizione (1): $(q, \cancel{\mathrm{a}}\mathrm{bb}, \cancel{B}) = (q, \mathrm{bb}, BB)$
   Produzione (2): $B \to \mathrm{a}B$
   Transizione (2): $(q, \cancel{\mathrm{a}}\mathrm{bb}, \cancel{S}B) = (q, \mathrm{bb}, BB)$

5. Poiché si sono ottenute due transizioni identiche, allora è possibile analizzarne solo una delle due (l'altra proseguirà in modo analogo).
   Quindi si legge il simbolo 'b' dal nastro e B dallo *stack*, generando una nuova biforcazione (non determinismo):

$$\begin{aligned}
&\text{Produzione (1): } B \to \mathrm{b} \\
&\text{Transizione (1): } (q, \not{\mathrm{b}}, \not{B}B) = (q, \mathrm{b}, B) \\
&\text{Produzione (2): } B \to \mathrm{b}S \\
&\text{Transizione (2): } (q, \not{\mathrm{b}}, \not{B}B) = (q, \mathrm{b}, SB)
\end{aligned}$$

6. Si prosegue per entrambe le ramificazioni leggendo lo stesso simbolo 'b' dal nastro e un simbolo diverso sullo *stack*:

$$\begin{aligned}
&\text{Produzione (1.1): } B \to \mathrm{b} \\
&\text{Transizione (1.1): } (q, \not{\mathrm{b}}, \not{B}) = (q, \varepsilon, \varepsilon) \quad \text{[Sì]} \\
&\text{Produzione (1.2): } B \to \mathrm{b} \\
&\text{Transizione (1.2): } (q, \not{\mathrm{b}}, \not{B}) = (q, \varepsilon, S) \quad \text{[No]} \\
&\quad\text{Produzione (2): } S \to \mathrm{b}A \\
&\quad\text{Transizione (2): } (q, \not{\mathrm{b}}, \not{S}B) = (q, \varepsilon, AB) \quad \text{[No]}
\end{aligned}$$

Poiché uno dei risultati ottenuti giunge a pila vuota, allora la stringa "aababb" è riconosciuta.

Lemma 3.6.5: Eliminazione della ricorsione sinistra

Sia $G = \langle V, T, S, P \rangle$ una grammatica CF (senza simboli inutili) e le A-produzioni sono tutte le produzioni che hanno il simbolo $A \in V$ a sinistra:

$$A \to \underbrace{A\alpha_1 | \dots | A\alpha_m}_{A\text{-produzioni}} | \beta_1 | \dots | \beta_m$$

Si costruisce $G' = \langle V \cup \{B\}, T, S, P' \rangle$, dove $\{B\}$ è un nuovo simbolo non-terminale, con l'insieme di nuove produzioni P':

$$P' = (P \setminus A\text{-produzioni}) \cup \left\{ \begin{array}{ll} A \to \beta_i | \beta_i B & \text{tale che } i \in \{1 \dots n\} \\ B \to \alpha_j | \alpha_j B & \text{tale che } j \in \{1 \dots m\} \end{array} \right\}$$

Questa costruzione permette che $\mathscr{L}(G) = \mathscr{L}(G')$.

Dimostrazione

Si dimostrano entrambe le inclusioni:

- **Prima inclusione** $\subseteq$: G è una grammatica CF e anche G' in FNG è CF.
- **Seconda inclusioni** $\supseteq$: si considera la seguente derivazione:

$$A \xrightarrow{G} A\alpha_{i_1} \xrightarrow{G} A\alpha_{i_2}\alpha_{i_1} \xrightarrow{G}{}^{*} x \in T^*$$

In questa sequenza, per arrivare a x, bisogna aver esaurito il simbolo A tramite una produzione $A \to \beta_j$. Ciò vuol dire che da A, con un certo numero di passi nella grammatica G si arriva a $\beta_j\alpha_{i_n} \dots \alpha_{i_2}\alpha_{i_1}$ con $h \geq 0$:

$$A \xrightarrow{G}{}^{*} \beta_j\alpha_{i_h} \dots \alpha_{i_2}\alpha_{i_1}, \quad h \geq 0$$

Allora ragionando all'interno di G' accade che da A, in G', si genera $\beta_j B$ e successivamente, sempre in G', si deriva $\beta_j\alpha_{i_h}B$.
Infine con un certo numero di passi in G' si arriva alla stringa $\beta_j\alpha_{i_h} \dots \alpha_{i_1}$:

$$A \xrightarrow{G'} \beta_j B \xrightarrow{G'} \beta_j\alpha_{i_h}B \xrightarrow{G'}{}^{*} \beta_j\alpha_{i_h} \dots \alpha_{i_1}$$

Ecco che si è derivata la medesima stringa finale sia con G sia con G'.

Esempio

Si suppone di avere la seguente grammatica:

$$A \to Aa|b$$

Questa grammatica genera il linguaggio ba^*:

$$b$$
$$ba$$
$$baa$$
$$baaa$$
$$\dots$$

$\triangleright$ Come si costruisce la grammatica eliminando la ricorsione sinistra?
Occorre eliminare le A-produzioni:

$$A \to Aa \quad \text{(presenta ricorsione)}$$
$$A \to b \quad \text{(non presenta ricorsione)}$$

Quindi si mantengono le produzioni senza ricorsione e si creano nuove produzioni, con quegli stessi simboli che non hanno ricorsione, aggiungendo un nuovo simbolo

non-terminale a destra:

$$A \to Aa|b$$
$$\downarrow$$
$$A \to b|bZ$$

A questo punto si creano delle produzioni per il nuovo simbolo non-terminale Z aventi i simboli delle A-produzioni:

$$Z \to a|aZ$$

In conclusione si è ricavata una nuova grammatica in FNG:

$$A \to b|bZ$$
$$Z \to a|aZ$$

Esempio

Si suppone di avere la seguente grammatica:

$$A \to Aa|Ab|b|c$$

Questa grammatica genera il linguaggio $\{b, c\} \cdot a, b^*$.

$\triangleright$ Come si costruisce la grammatica eliminando la ricorsione sinistra?
Nello stesso modo dell'esempio precedente: si eliminano le A-produzioni creandone di nuove con un simbolo non-terminale Z:

$$A \to b|c|bZ|cZ$$
$$Z \to a|b|aZ|bZ$$

Questa nuova grammatica è in FNG.

Osservazione

La FNG consiste in produzioni del tipo:

$$A \to \underbrace{a}_{\in T} \underbrace{z_1 \dots z_n}_{\in V}$$

Di conseguenza una produzione che presenta ricorsione sinistra non può essere in FNG.

Tuttavia seguendo questa procedura non si ottiene sempre una grammatica FNG. Infatti si suppone di avere la seguente grammatica:

$$A \to AB|BA|a$$
$$B \to b|c$$

In questo caso BA non causa problemi, in quanto nonostante ci siano due simboli non-terminali, non è una A-produzione. Dunque solo AB è un'A-produzione e bisogna eliminarla:

$$\begin{aligned} A &\to BA|a|BAZ|aZ \\ z &\to B|BZ \\ B &\to b|c \end{aligned}$$

Questa non è ancora in FNG, perché ci sono delle produzioni aventi simboli non-terminali a sinistra. Per passare in FNG basta soltanto rimpiazzare nelle produzioni le relative derivazioni di simboli terminali:

$$\begin{aligned} A &\to bA|cA|a|bAZ|cAX|aZ \\ z &\to b|c|bZ|cZ \end{aligned}$$

Questa grammatica è in FNG. Nel caso del linguaggio $\mathscr{L} = \{a^n b^n \mid n \geq 1\}$, con grammatica $S \to aSb|ab$, la sua FNG corrisponde a:

$$\begin{aligned} S &\to aSb|aB \\ B &\to b \end{aligned}$$

Teorema 3.6.6

Se il linguaggio $\mathscr{L}$ è CF, allora esiste un M ε-APND tale che $\mathscr{L} = \mathscr{L}_P(M)$, dove M legge a ogni passo un simbolo dal nastro (non effettua ε-transizioni).

Dimostrazione

Si deve costruire l'automa, per cui si suppone vero che $\mathscr{L}$ sia CF e di conseguenza $\mathscr{L} = \mathscr{L}(G)$, dove $G = \langle V, T, P, S \rangle$ in FNG. Quindi si definisce M avente:

- L'insieme degli stati $Q = \{q\}$.
- L'insieme dei simboli di pila $R = V$ (variabili non-terminali).
- L'alfabeto è $\Sigma = T$ (simboli terminali).
- Lo stato iniziale è l'unico presente $q_0 = q \in Q$.
- Il simbolo in testa alla pila è il simbolo iniziale della grammatica $z_0 = S$.
- L'insieme degli stati finali F è vuoto, poiché si accetta per pila vuota.

Le transizioni sono definite come:

$$(q, \alpha) \in t(q, a, A) \overset{\Delta}{\iff} A \to a\alpha \in P$$

Quindi la coppia, stato q e sequenza α sulla pila, appartiene alla transizione dello stato q leggendo a dal nastro e A (simbolo non-terminale) in testa alla pila. Questa costruzione della transizione dell'automa vale se e solo se la produzione per cui da A si passa ad $a\alpha$ appartiene a P.
Inoltre tale definizione implica che partendo da $(q, a, A\beta)$, con un solo passo in M, si va in $(q, \varepsilon, \alpha\beta)$ se e solo se $xA\beta$, in un passo in G, va in $xa\alpha\beta$:

$$(q, a, A\beta) \xrightarrow{M}{}^{1} (q, \varepsilon, \alpha\beta) \iff xA\beta \xrightarrow{G}{}^{1} xa\alpha\beta$$

Per induzione sul numero di passi in G, si dimostra che:

$$xA\beta \xrightarrow{G}{}^{n} xy\alpha\beta \iff (q, y, A\beta) \xrightarrow{M}{}^{n} (q, \varepsilon, \alpha\beta)$$

dove $|y| = n$. Quindi succede che tutte le stringhe generate nel linguaggio sono quelle derivate dal simbolo iniziale S:

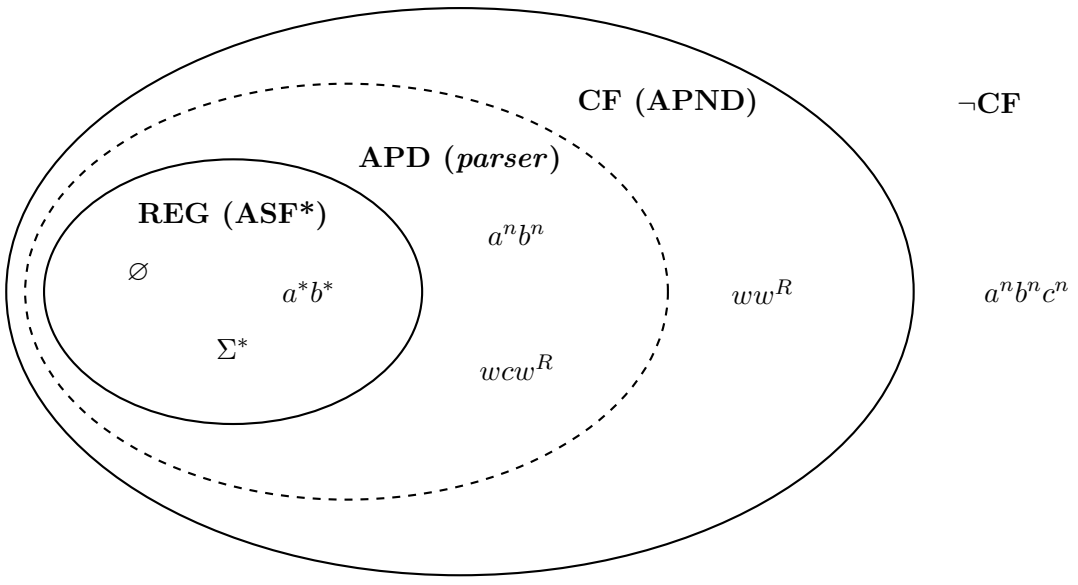
$$\begin{aligned} xA\beta \xrightarrow{G}{}^{n} xy\alpha\beta &\iff (q, y, A\beta) \xrightarrow{M}{}^{n} (q, \varepsilon, \alpha\beta)\varepsilon S\varepsilon \xrightarrow{G}{}^{*} \varepsilon \underset{\in T^*}{y} \varepsilon \\ &\iff (q, y, S) \xrightarrow{M}{}^{*} (q, \varepsilon, \varepsilon) \\ &\iff y \in \mathscr{L}(M) \end{aligned}$$

Un ε-ASFND è un caso particolare di un ε-APND. Da questo deriva che i linguaggi regolari siano un sottoinsieme dei linguaggi CF:

$$\text{REG} \subseteq \text{CF}$$

Nel dettaglio i linguaggi accettati da un ε-ASFND sono riconosciuti anche da un ASFND, dove quest'ultimo è un caso particolare di un APD. Ne consegue che i regolari sia contenuti nei linguaggi accettati dagli automi a pila deterministici:

$$\text{REG} \subseteq \text{Linguaggi APD}$$



La gerarchia di Chomsky continua poi con altri soprainsiemi più grandi. Per esempio $a^n b^n c^n$ appartiene ai **linguaggi dipendenti da contesto** (*content sensitive*, CS), nei quali gli automi usano nastri di memoria lunghi quanto la stringa in *input*.

Esempio: Linguaggio dipendente da contesto

Viene dato il linguaggio $\mathscr{L} = \{a^n b^n c^n \mid n > 0\}$ e l'alfabeto $\Sigma = \{a, b, c\}$. La sua grammatica corrisponde a:

$$
\begin{aligned}
S &\to abc|aAbc \\
Ab &\to bA \\
Ac &\to Bbcc \\
bB &\to Bb \\
aB &\to aa|aaA
\end{aligned}
$$

Come si può notare nella parte sinistra della produzione i simboli non-terminali sono contestualizzati da specifici simboli.

Nozione intuitiva di algoritmo e MdT

Si definisce un linguaggio $\mathscr{L}$, associato a una funzione f, come una sequenza di simboli $a^{f(n)}$, tale che $n \in \mathbb{N}$ e f sia una funzione che va dai numeri naturali ai numeri naturali:

$$\mathscr{L}_f = \{a^{f(n)} \mid n \in \mathbb{N}\}, \quad \Sigma = \{a\}$$
$$f : \mathbb{N} \to \mathbb{N}$$

Per affermare che una stringa appartenga a questo linguaggio occorre definire una classe di funzioni "intuitivamente calcolabili"[1]

4.1 Funzioni primitive ricorsive

La funzione di Fibonacci è un esempio di **funzione intuitivamente calcolabile**, poiché è costruita mettendo insieme gli elementi precedenti all'elemento n-esimo che bisogna costruire:

$$\begin{cases} f(0) = 1 \\ f(1) = 1 \\ f(n+2) = f(n+1) + f(n) \end{cases}$$

Se si vuole calcolare $f(2)$, allora il risultato è uguale alla somma di $f(1)$ più $f(0)$; se si deve calcolare $f(100)$, allora si somma $f(99)$ a $f(98)$. Generalizzando questo calcolo per ogni numero che si attribuisce a n, si ottiene un procedimento di regressione che termina in due casi base forniti nella definizione ricorsiva.

Un altro esempio di funzione intuitivamente calcolabile è il calcolo del fattoriale:

$$\begin{cases} !0 = 1 \\ !(x+1) = \ !x \cdot (x+1) \end{cases}$$

▷ Come si definiscono le **funzioni primitive ricorsive**?
Queste funzioni vengono costruite induttivamente: sono presenti casi base e successivamente una composizione di funzioni complesse tramite funzioni più semplici.

[1] Esiste un algoritmo intuitivo realizzato da un automa con una determinata memoria.

4.1.1 Funzioni ricorsive di base

Le **funzioni ricorsive di base**[2] sono:

- Funzione $\lambda x.0$; prende in *input* x e restituisce in *output* lo 0.

$$\lambda x.0 = f(x) = 0$$

- Funzione $\lambda x.x + 1$; prende in *input* un valore x e restituisce in *output* il suo successore $x + 1$.

$$\lambda x.x + 1 = f(x) = x + 1$$

- Funzione di proiezione $\lambda x_1 \dots x_i \dots x_n.x_i$; presi in *input* n valori di x, in *output* viene restituito il valore x_i.
  Un esempio di questa tipologia di funzioni è la funzione identità $\lambda x.x$.

Osservazione

Componendo queste prime due funzioni si generano tutti i numeri naturali.

4.1.2 Schemi di composizione

Gli **schemi di composizione** che definiscono le funzioni primitive ricorsive sono due. Si dice che $f : \mathbb{N}^n \to \mathbb{N}$ è definita per:

- **Composizione** da g e h se $f(x) = g(h(\overline{x}))$.
- **Ricorsione primitiva** da g e h se:

$$\begin{cases} f(0, \overline{x}) = g(\overline{x}) \\ f(n+1, x) = h(f(n, \overline{x}), n, \overline{x}) \end{cases}$$

Dove il primo argomento della funzione, definita per ricorsione primitiva, è l'indice della ricorsione e deve essere sempre presente, mentre i successivi parametri possono essere anche assenti.

Si afferma che quando l'indice di ricorsione è 0, allora il valore della funzione assume il caso base $g(\overline{x})$. Altrimenti se l'indice di ricorsione è $n + 1$ (maggiore stretto di 0), allora il valore della funzione è uguale all'applicazione di un'altra funzione h sull'indice di ricorsione precedente della funzione f stessa.

[2] La notazione λ esplicita il parametro della funzione, cioè prende in *input* ciò che sta prima del punto e restituisce in *output* ciò che sta dopo.

Definizione 4.1.1: Insieme delle funzioni primitive ricorsive PR

È la più piccola classe di funzioni che contiene le funzioni ricorsive di base ed è chiusa per composizione e ricorsione primitiva.

Teorema 4.1.1

PR sono funzioni totali: dato un qualunque numero, la funzione primitiva ricorsiva restituisce un valore.

Dimostrazione

Per induzione sulla struttura di PR:

- Funzioni ricorsive di base: sono totali. Per esempio $\lambda x.0$, $\lambda x.x+1$ e $\lambda x_1 \dots x_i \dots x_n.x_i$ sono sempre definite.
- Si suppone che $h, g \in \text{PR}$, allora per ipotesi induttiva h, g sono totali.
  - Composizione: se $\lambda x.g \circ h(x) = \lambda x.g(h(x))$, allora è totale.
    Perché la composizione di due funzioni totali (entrambe definite) risulta sempre definita.
  - Ricorsione primitiva: si definisce una relazione sulle coppie dei numeri:

$$\underbrace{(n, \overline{x}) \sqsubseteq (m, \overline{x})}_{\subseteq \mathbb{N} \times \mathbb{N}^k} \overset{\Delta}{\iff} m = n + 1$$

    Pertanto una coppia $(n, \overline{x})$ è più piccola di una coppia $(m, \overline{x})$ se e solo se $m = n+1$; dove $\overline{x}$ è un vettore di numeri. Questa relazione è ben fondata, cioè non esistono sequenze discendenti all'infinito di quella relazione. Ciò implica che le chiamate di f ricorsive siano:

$$(0, \overline{x}) \sqsubseteq (1, \overline{x}) \sqsubseteq (2, \overline{x}) \sqsubseteq \cdots \sqsubseteq (n, \overline{x})$$

    Essendo $(0, \overline{x}) = g$ totale, si passa alla coppia successiva tramite h (totale anch'essa per ipotesi) e si continua fino ad arrivare al risultato finale, di conseguenza f è totale.

Se $f \in \text{PR}$, allora esistono le funzioni $f_1 \dots f_n \in \text{PR}$ tali che:

- $f_n = f$, ovvero ogni funzione primitiva ricorsiva è ottenuta componendo un numero di funzioni più semplici.

- $\forall j \leq n.f_j$ è ricorsiva di base, oppure è ottenuta per schemi di composizione o ricorsione primitiva da f_i con $i < j$.

Dunque ogni funzione primitiva ricorsiva ha una successione di funzioni $f_1, \ldots, f_n$ che la definiscono; l'ultima di queste funzioni è esattamente la funzione f, mentre tutte le funzioni precedenti sono quelle che bisogna utilizzare per composizione (a partire dalle funzioni di base) per ottenere f_n.

Esempio

Vengono fornite le seguenti funzioni ricorsive di base:

$$f_1 = \lambda x.x$$
$$f_2 = \lambda x.x + 1$$
$$f_3 = \lambda x_1 x_2 x_3.x_2$$

Le funzioni f_4 e f_5 sono ottenute componendo alcune delle precedenti:

$$f_4 = f_2 \circ f_3$$
$$f_5 = \begin{cases} f_5(0, x) = f_1(x) \\ f_5(n+1, x) = f_4(n, f_5(n, x), x) \end{cases}$$

Si può notare che f_5 richiede la definizione di f_1 e f_4, la quale quest'ultima richiede la definizione di f_2 e f_3; dove f_1, f_2 e f_3 sono funzioni ricorsive di base. Di conseguenza la successione di funzioni, che porta dalle ricorsive di base fino a f_5, definisce la funzione f_5 stessa.

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

Le seguenti funzioni possono essere definite mediante ricorsione primitiva:

- **Somma**: $+(x, 0) = x$; se si somma nel caso base lo 0 alla x, allora si ottiene ancora la x (identità).
  $+(x, n+1) = (+(x, n)) + 1$; se la somma è generalizzata con $n+1$, allora è uguale al successore della somma (x, n) più un'unità (è ancora una ricorsiva di base del tipo $\lambda x.x + 1$).

Poiché questa è una funzione primitiva ricorsiva, definita per schema di ricorsione primitiva a partire dalle ricorsive di base identità e successore, allora $+ \in$ PR e la si può utilizzare dentro uno schema di ricorsione primitiva per definire il prodotto.

- **Moltiplicazione**: $\cdot(x, 0) = 0$; il prodotto tra x e 0 risulta 0 (ricorsiva di base $\lambda x.0$).
  $\cdot(x, n+1) = +(\cdot(x, n), x)$; il prodotto di x per $n+1$ impiega l'uso della funzione somma $+ \in$ PR, di conseguenza risulta essere uguale alla somma del prodotto (x, n) e x.

In questo modo la funzione di moltiplicazione è descritta come un'iterazione di somma e ne consegue che $\cdot \in$ PR.

- **Potenza**: $x^0 = 0+1 = 1$; l'elevazione di x alla 0 è composizione di $\lambda x.0$ e $\lambda x.x+1$.
  $x^{n+1} = \cdot(x^n, x)$; l'elevazione generalizzata di x alla $n+1$ corrisponde al prodotto di (x^n, x).

La funzione per calcolare la potenza include il prodotto che a sua volta include la somma, la quale include il successore e l'identità. Questa sequenza di funzioni consente di affermare che le potenze appartengono a PR.

- **Iperpotenza**: $\texttt{iper}(x, 0) = x$; il caso base consiste nell'identità ricorsiva primitiva.
  $\texttt{iper}(x, n+1) = \texttt{iper}(x, n)^x$; la generalizzazione dell'iperpotenza alla $n+1$ utilizza la potenza primitiva ricorsiva.

4.1.3 Adeguatezza di PR

▷ PR coincide con tutte le funzioni intuitivamente calcolabili?
No, PR non coincide con tutte le funzioni intuitivamente calcolabili. Ci sono due modi per verificalo: la seguente dimostrazione formale e l'esistenza di una funzione che esca fuori da PR.

Dimostrazione: Prima argomentazione

Si suppone che PR coincida, perciò l'alfabeto è composto da

$$\Sigma = \{a, b, c, \ldots, z, 1, \ldots, 9, =, (,)\}$$

e con le lettere si codificano le variabili. Se $f \in$ PR e si rappresenta con $[\![f(\overline{x})]\!] \in \Sigma^*$ la sequenza di simboli che esplicita il calcolo di $f(\overline{x})$, allora questo definisce una stringa.

▷ Qual è il linguaggio dato dalle stringhe che rappresentano il calcolo di funzioni primitive ricorsive, tale per cui la sequenza in *input* appartenga a $\mathbb{N}^k$?

$$\mathscr{L}_f = \{[\![f(\overline{x})]\!] \mid \overline{x} \in \mathbb{N}^k\}, \quad f \in \text{PR}$$

Si ricorda che se $f \in \text{PR}$, allora:

$$\exists f_1, \ldots, f_n \in \text{PR} : f = f_n$$
$$\forall j \leq n : \left[\begin{array}{l} f_j \text{ è base} \\ f_j \text{ è composizione di } f_i,\ i \leq j \end{array} \right.$$

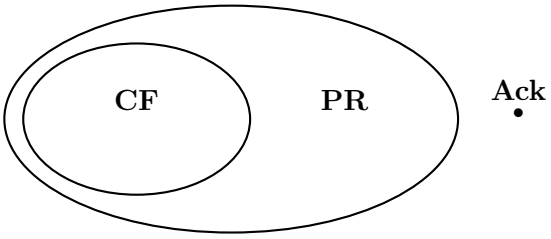
Questo implica che $|\text{PR}| = \omega$ e poiché è numerabile si può derivare una numerazione per le funzioni ricorsive: $f_0, f_1, f_2, \ldots, f_n, \cdots \in \text{PR}$. Dunque si definisce una funzione $h(x) = f_x(x) + 1$, la quale è intuitivamente calcolabile: si ha un procedimento intuitivo con cui si riesce a stabilire il calcolo. Tuttavia dato che si è supposto $h \in \text{PR}$, allora esiste un indice $n_0 \in \mathbb{N}$ tale che $h = f_{n_0}$.

$$f_{n_0} = (n_0) = h(n_0) = f_{n_0}(n_0) + 1$$

Questo è assurdo, perché se ogni funzione primitiva ricorsiva è totale - passando un numero naturale come parametro, la funzione restituisce un numero naturale -, allora non può esistere un numero naturale che sia uguale a sé stesso e al successore.

Funzione di Ackermann

▷ Quanto è grande l'insieme PR?



- **Ricorsive di base**:

$$\lambda x.0$$
$$\lambda x.x + 1$$
$$\lambda x_1, \ldots, x_i, \ldots, x_n.x_i$$

- **Composizione**:

$$f = h \circ g$$

- **Ricorsione primitiva**:

$$f(0, \overline{x}) = g(\overline{x})$$
$$f(n+1, \overline{x}) = h(f(n, \overline{x}), n, \overline{x})$$

Wilhelm Ackermann dimostra che esiste una funzione fuori dall'insieme PR che non è definibile mediante la **ricorsione primitiva**, nonostante sia una funzione totale e intuitivamente calcolabile.

Definizione 4.1.2: Funzione di Ackermann (1928)

La funzione è definita come:

$$\begin{cases} \texttt{Ack}(0, y) = y + 1 \\ \texttt{Ack}(x + 1, 0) = \texttt{Ack}(x, 1) \\ \texttt{Ack}(x + 1, y + 1) = \texttt{Ack}(x, \texttt{Ack}(x + 1, y)) \end{cases}$$

$\texttt{Ack}(x, \texttt{Ack}(x + 1, y))$ non utilizza uno schema di ricorsione primitiva, perché la chiamata all'interno è la stessa della chiamata esterna.

Scegliendo come parametri della funzione la coppia di valori $(4, 2)$ si ottiene:

$$\texttt{Ack}(4, 2) = 2^{2^{2^{2^2}}} - 3$$

Il risultato cresce in modo troppo elevato a partire già da indici piccoli.

Teorema 4.1.2

Ack non appartiene alle primitive ricorsive PR.

Dimostrazione: Seconda argomentazione

Si definisce un insieme $\mathscr{A}$ come la famiglia di funzioni f tale che sia strettamente minore di Ack:

$$\mathscr{A} = \{f \mid f < \texttt{Ack}\}$$

Ne consegue che $\texttt{Ack} \notin \mathscr{A}$. Dimostrando che $\text{PR} \subseteq \mathscr{A}$ implica che $\texttt{Ack} \notin \text{PR}$.
Per induzione si mostra che $\text{PR} \subseteq \mathscr{A}$, ovvero per ogni vettore $\overline{x}$ in *input* e per ogni funzione $f \in \text{PR}$, esistono due numeri $m, n \in \mathbb{N}$ tali che $f(\overline{x})$ sia strettamente minore di $\texttt{Ack}(m, n)$:

$$\forall \overline{x}. \forall f \in \text{PR}. \exists m, n \in \mathbb{N} : f(\overline{x}) < \texttt{Ack}(m, n)$$

dove m dipende dalla funzione f e $n = \texttt{max}(\overline{x})$; se $\overline{x} = x_1 \dots x_k$, allora $n = \texttt{max}(x_1 \dots x_k)$. Si considera che $\overline{x} \in \mathbb{N}^k$, con $k \in \mathbb{N}$.

- **Casi base**:
  - $\lambda x.0 = 0 < y + 1 = \texttt{Ack}(0, y)$. Questo implica che $\lambda x.0 \in \mathscr{A}$.

- $\lambda x.x + 1 = x + 1 < x + 2 = \texttt{Ack}(1, x)$. Infatti:

$$\begin{aligned}\texttt{Ack}(1, x) &= \texttt{Ack}(0, \texttt{Ack}(1, x - 1)) \\ &= \texttt{Ack}(1, x - 1) + 1 \\ &\vdots \\ &= \texttt{Ack}(1, 0) + x \\ &= \texttt{Ack}(0, 1) + x = x + 2\end{aligned}$$

  Questo implica che $\lambda x.x + 1 \in \mathscr{A}$.

- $\lambda x_1 \dots x_i \dots x_k.x_i \leq \underbrace{\texttt{max}(\overbrace{x_1 \dots x_k}^{\overline{x}})}_{\dot{x}} < \dot{x} + 1 = \texttt{Ack}(0, \dot{x})$.

  Questo implica che la funzione di proiezione appartiene:
  $\lambda x_1 \dots x_i \dots x_n.x_i \in \mathscr{A}$.

- **Passo induttivo**: si considerano $g_1 \dots g_m$ funzioni k-arie appartenenti a PR e h funzione m-aria in PR. Si suppongono vere due ipotesi induttive:

  1. $\forall i = 1 \dots m : g_i(\overline{x}) < \texttt{Ack}(r_i, \dot{x})$, dove r_i è associato alla funzione g_i e $\dot{x} = \texttt{max}(\overline{x})$.
  2. $h(y_1 \dots y_m) < \texttt{Ack}(s, \dot{y})$, dove s è associato alla funzione h e $\dot{y} = \texttt{max}(y_1 \dots y_m)$.

  - Composizione: a partire da queste ipotesi si dimostra che componendo le funzioni h con le funzioni g si ottiene ancora una funzione interna ad $\mathscr{A}$.
    - Si definisce la composizione $f(\overline{x}) \doteq h(g_1(\overline{x}), \dots, g_m(\overline{x}))$, dimostrando che $f \in \mathscr{A}$. Inoltre fissando un *input* $\overline{x} \in \mathbb{N}^k$ viene definita $g_j(\overline{x}) \doteq \texttt{max}(g_1(\overline{x}), \dots, g_m(\overline{x}))$.
    - Per le ipotesi induttive $f(\overline{x})$ è composta da tutte parti appartenenti ad $\mathscr{A}$.
    - Utilizzando l'ipotesi induttiva (2) si ottiene la seguente maggiorazione:
      $$f(\overline{x}) = h(g_1(\overline{x}) \dots g_m(\overline{x})) < \texttt{Ack}(s, g_j(\overline{x}))$$
      Sfruttando l'ipotesi induttiva (1) si ricava un'ulteriore maggiorazione:
      $$\texttt{Ack}(s, g_j(\overline{x})) < \texttt{Ack}(s, \texttt{Ack}(r_j, \dot{x}))$$

Per monotonia è possibile aumentare il valore $\dot{x}$:

$$\begin{aligned}\texttt{Ack}(s, \texttt{Ack}(r_j, \dot{x})) &\overset{\text{m}}{<} \texttt{Ack}(s, \texttt{Ack}(r_j, \dot{x}+1)) \\ &\overset{\text{m}}{<} \texttt{Ack}(s+r_j+1, \texttt{Ack}(s+r_j+2, \dot{x}+1))\end{aligned}$$

Queste operazioni sono state svolte per riuscire a ricondurre il primo parametro a una x e il primo parametro del secondo parametro a una $x+1$:

$$\texttt{Ack}(\underbrace{s+r_j+1}_{x}, \texttt{Ack}(\underbrace{s+r_j+2}_{x+1}, \dot{x}+1))$$

Così facendo è possibile applicare lo schema di ricorsione definito per la funzione $\texttt{Ack}$. Poiché $\texttt{Ack}(x+1, y+1) = \texttt{Ack}(x, \texttt{Ack}(x+1, y))$, allora si effettua la sostituzione:

$$\texttt{Ack}(s+r_j+1, \texttt{Ack}(s+r_j+2, \dot{x}+1)) = \texttt{Ack}(s+r_j+2, \dot{x})$$

Ne deriva che $f \in \mathscr{A}$, perché è stata maggiorata da una funzione che ha come argomenti una composizione, tra h e g_j, e il valore massimo $\dot{x}$.

– Ricorsione primitiva: sia g una funzione k-aria e h una funzione $k+2$-aria, con $g, h \in \mathscr{A}$. Si hanno due ipotesi induttive:

1. $g(\overline{x}) < \texttt{Ack}(r, \dot{x})$, con $\dot{x} = \texttt{max}(\overline{x})$.
2. $h(y_1, \ldots, y_k, y_{k+1}, y_{k+2}) < \texttt{Ack}(s, \dot{y})$, con $\dot{y} = \texttt{max}(y_1, \ldots, y_{k+2})$.

Occorre dimostrare che $f(\overline{x}, n) \in \mathscr{A}$ per ogni $n \in \mathbb{N}$. Sia $q \doteq \texttt{max}(r, s)+1$, allora per induzione su $n \in \mathbb{N}$ si dimostra che:

$$f(\overline{x}, n) < \texttt{ack}(q, n + \texttt{max}(\overline{x}))$$

∗ **Caso base**: con $n = 0$ per definizione si ha:

$$f(\overline{x}, 0) = g(\overline{x}) < \texttt{Ack}(r, \dot{x})$$

La quale per monotonia è maggiorata da q:

$$\texttt{Ack}(r, \dot{x}) < \texttt{Ack}(q, \dot{x}+0)$$

∗ **Passo induttivo**: si assume l'ipotesi sulla chiamata ricorsiva $f(\overline{x}, n) < \texttt{Ack}(q, n+\dot{x}))$. Considerando il caso $n+1$ si ottiene:

$$f(\overline{x}, n+1) = h(\overline{x}, f(\overline{x}, n), n)$$

Applicando l'ipotesi induttiva (2) si maggiora la funzione h:

$$h(\overline{x}, f(\overline{x}, n), n) < \mathtt{Ack}(s, z)$$

dove $z = \mathtt{max}(\overline{x}, n, f(\overline{x}, n))$.
Poiché $\mathtt{max}(\overline{x}, n) \leq n + \mathtt{max}(\overline{x}) = n + \dot{x} < \mathtt{Ack}(q, n + \dot{x})$ e per ipotesi induttiva è noto che $f(\overline{x}, n) < \mathtt{Ack}(q, n + \dot{x})$, allora $z < \mathtt{Ack}(q, n + \dot{x})$.
Da questo si deriva che:

$$\begin{aligned} f(\overline{x}, n+1) &< \mathtt{Ack}(s, z) \\ &\overset{\mathrm{m}}{<} \mathtt{Ack}(s, \mathtt{Ack}(q, n + \dot{x})) \\ &\overset{\mathrm{m}}{<} \mathtt{Ack}(q-1, \mathtt{Ack}(q, n + \dot{x})) \\ &\doteq \mathtt{Ack}(q, n + 1 + \dot{x}) \end{aligned}$$

Ciò implica che $f(\overline{x}, n+1) < \mathtt{Ack}(q, n + 1 + \dot{x})$ e per induzione su n si deriva che:

$$\forall n \in \mathbb{N}. f(\overline{x}, n) < \mathtt{Ack}(q + \dot{x})$$

Quindi questa dimostrazione per induzione lega il l'argomento q alla funzione f e l'argomento $n + \dot{x}$ con la dimensione di n:

$$f(\overline{x}, n) < \mathtt{Ack}(q, n + \dot{x})$$

Per concludere la dimostrazione di $f \in \mathscr{A}$, è necessario mostrare che si mantiene lo schema: per il secondo argomento della maggiorazione di $\mathtt{Ack}$, si ha il massimo degli argomenti della funzione f.
Si considera $z = \mathtt{max}(\overline{x}, n) = \mathtt{max}(x_1 \dots x_k, n)$. Per la precedente proprietà dimostrata con l'induzione, si ha che:

$$\begin{aligned} f(\overline{x}, n) &< \mathtt{Ack}(q, n + \dot{x}) \\ &\overset{\mathrm{m}}{\leq} \mathtt{Ack}(q, 2z) \\ &< \mathtt{Ack}(q, 2z + 3), \quad \text{poiché } \mathtt{Ack}(2, z) = 2z + 3 \\ &= \mathtt{Ack}(q, \mathtt{Ack}(2, z)) \\ &\overset{\mathrm{m}}{<} \mathtt{Ack}(q + 1, \mathtt{Ack}(q + 2, z + 1)) \\ &= \mathtt{Ack}(q + 4, z) \end{aligned}$$

Da ciò consegue che $f \in \mathscr{A}$, perciò $\mathtt{Ack} \notin \mathrm{PR}$.

4.2 Macchina di Turing

4.2.1 Concetto di algoritmo

Un **algoritmo** è definito in funzione delle seguenti caratteristiche fondamentali:

- sequenza finita di istruzioni (il programma);
- esiste un agente (automa) di calcolo autonomo;
- il calcolo avviene come sequenza discreta di passi elementari. Si inizia da uno stato iniziale S_0 e si va in una sequenza di stati successivi: $S_0 \to S_1 \to \cdots \to S_n \to \ldots$;
- l'*input* non ha un limite a patto che sia finito;
- la memoria non ha limiti;
- ogni singolo passo dipende da una quantità finita di informazioni, cioè nella serie di passaggi $S_0 \to S_1 \to \cdots \to S_n \to S_{n+1}$, il passaggio S_{n+1} dipende dalla qualità di informazioni presente da S_0 a S_n.
- il tempo a disposizione per il calcolo non ha limite;
- l'energia a disposizione per il calcolo non ha limite;
- sono permesse computazioni che non terminano.

Data questa analisi, l'ultimo punto è quello più importante al fine di non rendere paradossale il sistema.

Dimostrazione

Si suppone che gli automi di Turing (MdT) terminano sempre con una soluzione. Se si dispone di un alfabeto Σ, allora la macchina M di Turing è definita da un programma, con $M \in \Sigma^*$.

$\triangleright$ Quante sono le macchine di Turing?
Se $|\Sigma^*| = \omega = |\mathbb{N}|$, allora le MdT sono numerabili: $M_0, M_1, M_2, \ldots, M_n, \ldots$. Si costruisce una macchina M, che prende come *input* un numero $x \in \mathbb{N}$, e le si assegna una macchina x-esima più 1:

$$M(x) = M_x(x) + 1$$

Poiché l'ipotesi di partenza supponeva che le MdT terminassero sempre con una soluzione, allora il risultato è ancora un numero in $\mathbb{N}$. Quindi questa macchina M appena definita sta da qualche parte all'interno della numerazione delle macchine, con un certo indice x_0.
Se si prende la macchina M e come parametro si usa x_0, allora questa corrisponde

a $M_{x_0}(x_0)$, ma per la sua precedente definizione è uguale a $M_{x_0}(x_0) + 1$:

$$M(x_0) = M_{x_0}(x_0) = M_{x_0}(x_0) + 1$$

Tuttavia questo è un assurdo, perché non può essere uguale a sé stessa e al suo successore.

La supposizione di Turing, per la quale sono permesse computazioni che non terminano, rompe questo paradosso:

$$M(x_0) = \underbrace{M_{x_0}(x_0)}_{\infty} = \underbrace{M_{x_0}(x_0) + 1}_{\infty} \quad \rightarrow \quad \infty = \infty$$

4.2.2 Definizione della MdT

La **macchina di Turing** (MdT) M è definita come un automa avente:

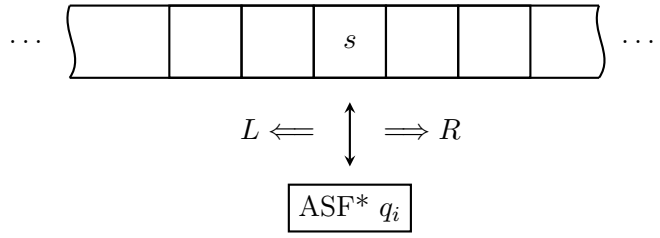
$$M = \langle Q, \Sigma, q_0, s_0, F \rangle$$

- Q è l'insieme di stati presenti;
- Σ è l'alfabeto utilizzato in memoria e lettura;
- q_0 è lo stato iniziale della macchina;
- s_0 è il simbolo iniziale, appartenente all'alfabeto, letto dal nastro;
- F è l'insieme di stati finali.

Gli stati della macchina sono finiti $|Q| < \omega$ e gli stati finali sono un sottoinsieme di quelli presenti $F \subseteq Q$. Inoltre anche l'alfabeto è finito $|\Sigma| < \omega$.

Dal punto di vista modellistico la macchina è composta da:

- un nastro illimitato in entrambe le direzioni (equivalente a esserlo anche per una sola direzione);
- il nastro è suddiviso in unità di memoria. Per ciascuna unità è presente un simbolo appartenente a Σ;
- il controllo è un automa a stati finiti di un certo tipo (ASF*) e posizionato in uno stato $q_i \in Q$.



La macchina può leggere e scrivere sul nastro in una qualunque posizione. Inoltre come operazione unitaria può spostarsi a sinistra o a destra.

▷ Come si descrive un'azione elementare della macchina?
Se la MdT si trova in uno stato q e legge un simbolo s, allora lo si può sostituire con un altro simbolo e spostare il controllore a destra o sinistra di un'unità. Il programma P è dato da una matrice avente come colonne tutti i simboli che si possono leggere in Σ, mentre come righe tutti gli stati $q_i \in Q$. Nelle celle di corrispondenza tra un simbolo s_j e uno stato q_i, è presente una tripla di informazioni suddivisa come: simbolo s_k da sovrascrivere a s_j, stato q_h in cui effettuare la transizione e infine il tipo di spostamento $X \in \{L, R\}$.

	$\dots$	s_j	$\dots$
$\vdots$			
q_i		$s_k q_h X$	
$\vdots$			

L'istruzione elementare di una MdT è composta da cinque elementi: $q_i s_j s_k q_h x$.

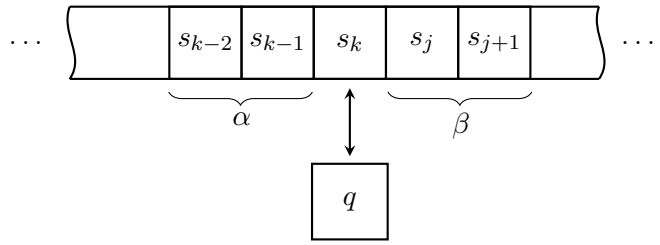
- Se la cella $(s_j q_i)$ è vuota e $q_i \notin F$, allora la MdT M termina e non accetta l'*input*.
- Se la cella $(s_j q_i)$ è vuota e $q_i \in F$, allora la MdT M termina e accetta l'*input*.

Definizione 4.2.3: Descrizione istantanea

Data una MdT M, la descrizione istantanea corrisponde a:

$$\mathrm{ID}_M\colon\ \alpha q s_k \beta \in \Sigma^* \times Q \times \Sigma^*$$

con $\alpha, \beta \in \Sigma^*$, $s_k \in \Sigma$ e $q \in Q$.



▷ Come vengono effettuati i calcoli nella MdT?
Si suppone di avere come descrizione istantanea $\delta qs\eta$, dove ciascun elemento corrisponde a:

- δ: sequenza precedente al simbolo s;
- q: stato corrente della macchina;
- s: simbolo da leggere sul nastro;
- η: sequenza successiva al simbolo s.

$$\begin{aligned} \delta qs\eta &\xrightarrow{M} \delta s'q'\eta && \text{se } qss'q'R \in M \\ \delta s''qs\eta &\xrightarrow{M} \delta q's''s'\eta && \text{se } qss'q'L \in M \end{aligned}$$

La prima operazione corrisponde a uno spostamento verso destra sul nastro, mentre la seconda operazione consiste in uno spostamento verso sinistra.

Definizione 4.2.4: Sequenza ammessa come descrizione istantanea

La sequenza α, appartenente alle descrizioni istantanee della macchina M, è terminale se per ogni sequenza β, appartenente alle descrizioni istantanee della macchina M, non si può andare da α in β con M:

$$\alpha \in \mathrm{ID}_M \Rightarrow \forall \beta \in \mathrm{ID}_M : \alpha \not\xrightarrow{M} \beta$$

Definizione 4.2.5: Computazione di una MdT

Una computazione è una sequenza arbitrariamente lunga (eventualmente infinita) di descrizioni istantanee:

$$\alpha_0 \xrightarrow{M} \alpha_1 \xrightarrow{M} \alpha_2 \xrightarrow{M} \ldots \xrightarrow{M} \alpha_n \xrightarrow{M} \ldots$$

- Una computazione termina se esiste una sequenza $\alpha_n \in \mathrm{ID}_M$ tale per cui:

  $$\alpha_0 \xrightarrow{M} \alpha_n \not\longrightarrow$$

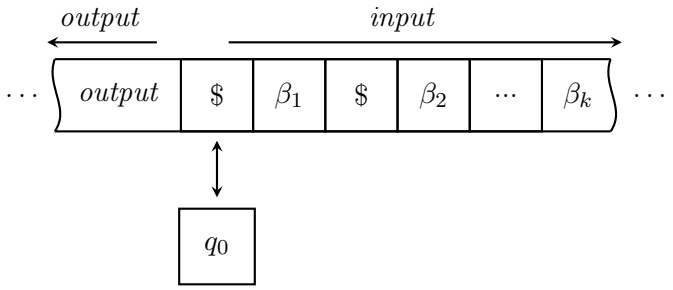
  dove α_n è terminale.

- Una computazione diverge se per ogni $n \in \mathbb{N}$ si ha sempre una transizione possibile in una configurazione che non sia finale:

  $$\alpha_n \xrightarrow{M} \alpha_{n+1}$$

4.2.3 Funzioni Turing calcolabili

Si definisce una codifica degli argomenti e una funzione $f : \mathbb{N}^k \to \mathbb{N}$, ossia $f(x_1, \ldots, x_k)$ con $x_i = \beta_i \in \Sigma^*$ codifica dell'*input*.



Definizione 4.2.6: Funzione Turing calcolabile

Una funzione $f : \mathbb{N}^k \to \mathbb{N}$ è Turing calcolabile se esiste una MdT M tale che, per ogni *input* $x_1, \ldots, x_n \in \mathbb{N}$, si ha:

- Se $f(x_1, \ldots, x_k)\downarrow$, allora:

$$\underbrace{q_0\$x_1\$\ldots\$x_k}_{\mathrm{ID}_0} \xrightarrow{M}{}^{*} f(x_1 \ldots x_k)q_f\$x_1\$\ldots\$x_k$$

  con $q_f \in F_M$.

- Se $f(x_1, \ldots, x_k)\uparrow$, allora:

$$q_0\$x_1\$\ldots\$x_k \xrightarrow{M}{}^{*} \alpha_n$$

  per ogni $n \in \mathbb{N}$ e $\alpha_n \neq \alpha_{n+1}$.

Si indica con il simbolo $\downarrow$ che il programma termina, mentre con $\uparrow$ il fatto che non termini mai.

Da questa definizione seguono svariati risultati, tra cui:

- Teorema di Shannon (1956): una MdT con n simboli ($|\Sigma| = n$) e m stati ($|Q| = m$) può essere simulata da una MdT con 2 stati, facendo aumentare Σ, oppure 2 simboli, aumentando Q.
- Una MdT con n nastri può essere simulata da una MdT con un solo nastro.
- Un automa a 2 pile è equivalente a una MdT.

4.2.4 Funzioni parziali ricorsive

Le funzioni primitive ricorsive sono un sottoinsieme stretto delle **funzioni parziali ricorsive**:

$$\text{PR} \subset \text{KR} = \text{MdT}$$

Definizione 4.2.7: Insieme delle funzioni parziali ricorsive KR

KR è la minima classe di funzioni che contiene PR ed è chiusa per minimizzazione.

La **minimizzazione** (o ricerca lineare incerta) è descritta come:

$$f : \mathbb{N}^{n+1} \to \mathbb{N} \text{ totale} \Rightarrow \varphi : \mathbb{N}^n \to \mathbb{N}$$

dove quest'ultima è definita se:

$$\varphi(\underbrace{x_1, \ldots, x_n}_{n}) = \mu z.f(\underbrace{x_1, \ldots, x_n, z}_{n+1})$$

Fissati i valori dei parametri $x_1, \ldots, x_n$, non è certo che la funzione abbia uno 0, pertanto si cerca il minimo[3] z che porti la funzioni a 0. Algoritmicamente significa:

$$\mu z.f(x_1, \ldots, x_n, z) = \begin{cases} \underbrace{\texttt{min}\{z \mid f(x_1, \ldots, x_n, z) = 0\}}_{A} & \text{se } A \neq \emptyset \\ \uparrow & \text{se } A = \emptyset \end{cases}$$

```
input(x1,...,xn)
z = 0
while f(x1,...,xn,z) != 0 then z++
output(z)
```

Si può affermare che la funzione avente come parametri $x_1, \ldots, x_n$ termina se e solo se l'insieme della z, tale che $f(x_1, \ldots, x_n, z)$ sia uguale a 0, è diverso dal vuoto:

$$\varphi(x_1, \ldots, x_n)\downarrow \iff \{z \mid f(x_1, \ldots, x_n, z) = 0\} \neq \emptyset$$

Esempio

Il risultato dell'espressione $\lfloor \log_a(x) \rfloor$ corrisponde al minimo y tale che l'operazione $\texttt{leq}(x, a^{y+1})$ sia uguale a 0:

$$\lfloor \log_a(x) \rfloor = \mu y.\texttt{leq}(x, a^{y+1}) = 0$$

[3] Il concetto di minimo viene espresso con il simbolo μ davanti alla variabile.

L'operazione `leq` $\in$ PR è definita nella seguente maniera:

$$\begin{cases} \texttt{leq}(x+1,0) = 1 \\ \texttt{leq}(0,0) = 1 \\ \texttt{leq}(0,x+1) = 0 \\ \texttt{leq}(x+1,y+1) = \texttt{x,y} \end{cases}$$
$$\texttt{leq}(x,y) = 0 \iff x < y$$

Lo schema delle funzioni parziali ricorsive di Kleene è robusto ed equivalente alle MdT.

Teorema 4.2.3: Robustezza di KR

Una funzione f appartiene a KR se e solo se f è Turing calcolabile:

$$f \in \text{KR} \iff f \text{ è Turing calcolabile}$$

Dimostrazione

Si dimostrano entrambe le implicazioni:

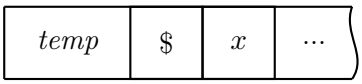
- **Prima implicazione** $\Rightarrow$: sia $f \in$ KR. Allora si costruisce una MdT che calcola f. Si applica l'induzione:
  - Caso base:

    $$\lambda x.0$$
    $$\lambda x.x+1$$
    $$\lambda x_1, \ldots, x_i, \ldots x_n.x_i$$

    Queste sono le ricorsive di base, ovvie in MdT.
  - Passo induttivo: siano h, g funzioni in KR e M_h, M_g le rispettive MdT, le quali esistono per ipotesi induttiva.

1. **Composizione** $M_{g \circ h}$:

```
input(x)
temp = M_h(x)
M_g(temp)
```

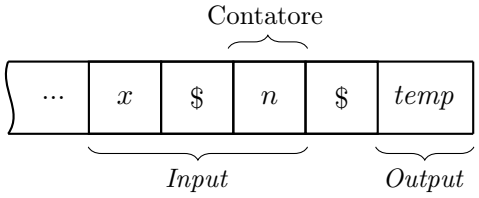


Dunque il risultato presente nella regione di memoria a destra viene copiato all'interno di *temp* (a sinistra).

2. **Ricorsione primitiva**:

$$\begin{cases} f(x,0) = g(x) \\ f(x,n+1) = h(n, f(x,n), x) \end{cases}$$

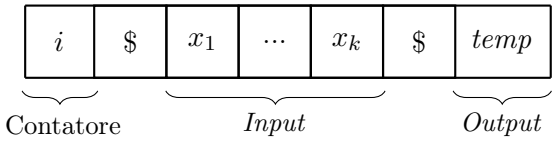
```
input(x)
input(n)
temp = M_g(x)
for i = 0 to n {
  temp = M_h(i, temp, x)
}
```



3. **Minimizzazione**:

$$\varphi(x_1, \ldots, x_k) = \mu z.f(x_1, \ldots, x_k, z) = 0$$

```
input(x1,...,xk)
i = 0
temp = M_f(x1,...,xk,i)
while temp != 0 {
  i++
  temp = M_f(x1,...,xk,i)
}
```

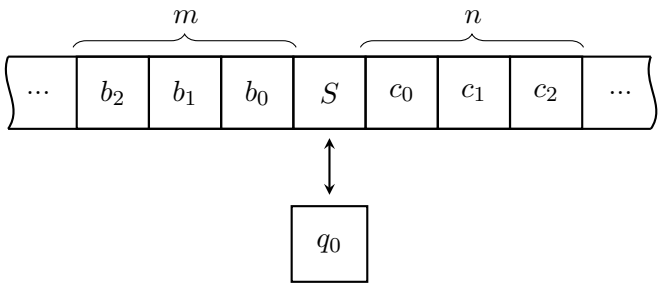


- **Seconda implicazione** $\Leftarrow$: siano $M \in$ MdT e (l'*input*) $x \in \mathbb{N}$.

$$M = \langle Q, \Sigma, q_0, s_0, F, P \rangle$$

  - $\Sigma = \{0, 1\}$.
  - $Q = \{q_0, \ldots, q_k\}$, con $k + 1$ stati e simboli.
  - $F = \{q_k\}$.

  Bisogna definire il calcolo di $M(x) \in$ MdT e passare alla sua implementazione con la funzione parziale ricorsiva $\varphi_M(x) \in$ KR.
  A ogni istante del calcolo la MdT ha una quantità finita di informazioni:



  Una descrizione istantanea consiste in:

  - uno stato $q \in [0, k]$;
  - un simbolo $s \in \{0, 1\}$;
  - il numero $m \in \mathbb{N}$ tale che:

$$m = \sum_i b_i \cdot 2^i$$

  - il numero $n \in \mathbb{N}$ tale che:

$$n = \sum_i c_i \cdot 2^i$$

  di conseguenza l'ID della macchina è una quadrupla di numeri:

$$(q, s, m, n) \in \mathbb{N}^4$$

Una descrizione istantanea viene trasformata in un'altra descrizione istantanea mediante funzioni di transizione che trasformano ciascun elemento della quadrupla in altri elementi:

$$(q, s, m, n) \xrightarrow{M} (q', s', m', n')$$

Poiché la macchina M è finita, allora si costruiscono le seguenti funzioni:

$$\begin{array}{ll} \mathbb{Q} : Q \times \Sigma \to Q & \text{(transizione di stato)} \\ \mathbb{S} : Q \times \Sigma \to \Sigma & \text{(transizione di simbolo)} \\ \mathbb{X} : Q \times \Sigma \to \underbrace{\{0, 1\}}_{L,R} & \text{(transizione di movimento)} \end{array}$$

Con queste funzioni si formula il primo passo della MdT:

$$\begin{aligned} q' &\doteq \mathbb{Q}(q, s) \\ s' &\doteq \begin{cases} n \texttt{ mod } 2 & \mathbb{X}(q, s) = 1 \\ m \texttt{ mod } 2 & \mathbb{X}(q, s) = 0 \end{cases} \\ m' &\doteq \begin{cases} 2m + \mathbb{S}(q, s) & \mathbb{X}(q, s) = 1 \\ m \texttt{ div } 2 & \mathbb{X}(q, s) = 0 \end{cases} \\ n' &\doteq \begin{cases} n \texttt{ div } 2 & \mathbb{X}(q, s) = 1 \\ 2n + \mathbb{S}(q, s) & \mathbb{X}(q, s) = 0 \end{cases} \end{aligned}$$

Si compongono in un'unica formula entrambi i casi:

$$\begin{aligned} q' &\doteq \mathbb{Q}(q, s) \\ s' &\doteq (m \texttt{ mod } 2) \cdot (1 - \mathbb{X}(q, s) + (n \texttt{ mod } 2) \cdot \mathbb{X}(q, s)) \\ m' &\doteq (2m + \mathbb{S}(q, s)) \cdot \mathbb{X}(q, s) + (m \texttt{ div } 2) \cdot (1 - \mathbb{X}(q, s)) \\ n' &\doteq (2n + \mathbb{S}(q, s)) \cdot (1 - \mathbb{X}(q, s)) + (n \texttt{ div } 2) \cdot \mathbb{X}(q, s) \end{aligned}$$

A questo punto non si conosce il numero di passi che bisogna eseguire per terminare la MdT, perché potrebbe non concludere mai. Di conseguenza si cerca il primo stato finale che la macchina raggiunge (se esiste) ed è un caso di minimizzazione.
Si prende la variabile $t \in \mathbb{N}$, che rappresenta i passi di calcolo, e la seguente funzione:

$$P_q \doteq \begin{cases} P_q(0, q, s, m, n) = q \\ P_q(t + 1, q, s, m, n) = P_q(t, q', s', m', n') \end{cases}$$

Questa funzione appartiene a PR. Se P_q è lo stato dopo t passi da (q, s, m, n), allora si definisce che $P_o(t, q, s, m, n)$ sia l'*output*, ossia il contenuto del nastro

a destra del simbolo in lettura dopo t passi a partire da (q, s, m, n):

$$\begin{cases} P_o(0, q, s, m, n) = n \\ P_o(t+1, q, s, m, n) = P_o(t, q', s', m', n') \end{cases}$$

Sapendo che $F = \{q_k\}$, la funzione calcolata dalla MdT secondo questa aritmetizzazione corrisponde a:

$$\varphi(x) = P_o(\overbrace{\mu t.P_q(t, x \texttt{ mod } 2, x, 0)}^{\text{passi per raggiungere } q_k}, 0, x \texttt{ mod } 2, x, 0)$$

Ne deriva che $\varphi \in$ KR implementa la funzione $\lambda x.M(x)$.

Corollario 4.2.1: Forma normale di Kleene

Ogni funzione Turing calcolabile può essere espressa nella forma:

$$\varphi(x) = C(\mu t.Z(t, x) = 0)$$

dove C e Z sono primitive ricorsive.

4.2.5 Tesi di Church-Turing

Questa tesi, non dimostrabile in senso assoluto, afferma che ogni funzione intuitivamente calcolabile è calcolabile da una macchina di Turing.

Tesi di Church-Turing (1936)

La classe delle funzioni intuitivamente calcolabili coincide con la classe delle funzioni Turing calcolabili.

Se φ è una funzione parziale ricorsiva (KR) e M è la MdT che calcola φ, allora si scrive come φ_M la funzione parziale ricorsiva calcolata da M.

▷ Come si rappresenta una funzione calcolata da una macchina?
Si associa la macchina alla funzione: φ_M, dove φ è la funzione e M è la macchina (il programma) che la calcola.

▷ Perché sono tutti equivalenti i linguaggi di programmazione?
Se si considera la classe delle funzioni $\{\varphi_P \mid P \in \text{Java}\}$, cioè le funzioni che sono descritte mediante il linguaggio Java, allora per la tesi di Church-Turing questo è equivalente alla classe delle MdT:

$$\{\varphi_P \mid P \in \text{Java}\} = \{\varphi_M \mid M \in \text{MdT}\}$$

A sua volta anche la classe di funzioni descritte dal C++ è uguale alla classe di funzioni descritte dalla MdT:

$$\{\varphi_P \mid P \in \text{C++}\} = \{\varphi_M \mid M \in \text{MdT}\}$$

Tale concetto si estende a tutti i linguaggi di programmazione.

> Un linguaggio di programmazione serve per esprimere facilmente alcune classi di funzioni.

Ne consegue che se $\varphi : \mathbb{N}^k \to \mathbb{N}$ è intuitivamente calcolabile, allora esiste una MdT enumerata da un certo $x \in \mathbb{N}$ tale che la funzione φ è calcolata da quella macchina:

$$\varphi : \mathbb{N}^k \to \mathbb{N} \Rightarrow \exists x \in \mathbb{N} : \varphi = \varphi_x$$

$\triangleright$ Come si associa un numero a una MdT?
Una MdT è definita come:

$$M = \langle \Sigma, Q, q_0, s_0, F, P \rangle$$

Il programma P definisce il comportamento della macchina. Se si aritmetizza P, allora si riesce ad aritmetizzare la macchina.
Nel dettaglio P è una stringa composta da sequenze finite di quintuple:

$$P \in (\Sigma \times Q \times \Sigma \times Q \times \{L, R\})^*$$

All'interno sono presenti tutti insiemi di cardinalità finita. Se si uniscono questi insiemi, allora la cardinalità totale è finita:

$$|\Sigma \cup Q \cup \{L, R\}| = n < \omega$$

Dunque si prendono i primi n numeri dispari $a_1, \ldots, a_n \in 2\mathbb{N} + 1$. È evidente che per ogni i, j ciascun valore è diverso:

$$\forall i, j : a_i \neq a_j$$

Allora il *Gödel number*[4] associato al programma P è il prodotto, che va da $i = 1$ fino $|P|$ (lunghezza del programma), di P_i con il simbolo corrispondente $a_{s(i)}$:

$$\mathtt{gn}(P) = \prod_{i=1}^{|P|} P_i^{a_{s(i)}}$$

[4]Una numerazione di Gödel è una funzione che assegna a ciascuna produzione di un linguaggio formale un unico numero naturale chiamato numero di Gödel.

Questa notazione ha come significato:

- P_i: è l'i-esimo numero primo;
- $s(i)$: è l'indice del simbolo i-esimo del programma.

Ecco che la funzione calcolata dalla MdT M corrisponde alla funzione calcolata avente come numero $\texttt{gn}(P)$:

$$\varphi_M = \varphi_{\texttt{gn}(P)}$$

Il punto interessante è che se si ha un numero naturale, allora lo si può **fattorizzare**:

$$x \in \mathbb{N} \Rightarrow \texttt{fatt}(x) = P_1^{x_1} \dots P_h^{x_h}$$

Si guardano i corrispondenti valori per $x_1 \dots x_h$, se questi valori sono numeri interi compresi tra 0 e n, allora si estraggono i simboli e si ricostruisce il programma da quegli esponenti. Tuttavia se questi numeri non corrispondono ai numeri dispari, con i quali si sono codificati i simboli, allora x non è un programma e lo si scarta.
Quindi da tutto ciò si deriva che:

- Se il programma è definito, allora si estrae il numero associato:

$$\forall i = 1 \dots h : x_i = a_j, \quad j \in [1, n] \Rightarrow P = x_1 \dots x_n$$

- Se il programma non è definito, cioè non c'è alcun numero associabile, allora si definisce un programma che non finisce mai:

$$P \equiv \texttt{while true \{ x = x \}}$$

▷ Di cosa ha bisogno un linguaggio di programmazione?
Se si ha un funzione parziale φ calcolabile, allora esiste una MdT. Un linguaggio di programmazione necessita soltanto delle seguenti operazioni:

0	Programma che restituisce 0
+1	Programma che restituisce il successore
skip	Programma che non fa niente
$\circ$	Composizione
`while`	Ciclo per iterare istruzioni

▷ Quante MdT esistono?
Le MdT sono tante quanti i programmi, ossia tanti quanti i numeri naturali; le MdT sono numerabili. Di conseguenza le funzioni calcolabili di Turing sono una piccola minoranza di tutte le possibili funzioni che si possono costruire.

$$|\{\varphi_P \mid P \in \mathscr{L}\}| < |\{\varphi \mid \varphi : \mathbb{N}^k \to \mathbb{N}\}|$$

Metaprogrammazione (universalità)

L'**universalità** delle MdT (dei sistemi di calcolo) è una proprietà fondamentale. Un sistema di calcolo ha la potenza delle MdT se possiede la caratteristica di essere un sistema che permette la **metaprogrammazione**.

> Nel momento in cui si giunge alla MdT, ovvero ai linguaggi di programmazione, si ottiene l'universalità: si permette la metaprogrammazione: si permette l'esistenza di un interprete.

Quindi esiste un programma (MdT), detto **inteprete** INT, tale che la funzione calcolata da questo programma prende $n+1$ argomenti e restituisce un numero:

$$\varphi_{\text{INT}} : \underbrace{\mathbb{N} \times \mathbb{N}^n}_{n+1} \to \mathbb{N}$$

È definita nella seguente maniera:

$$\varphi_{\text{INT}}(\underbrace{x, \overline{y}}_{n+1}) = \begin{cases} \varphi_x(\overline{y}) & \text{se } \varphi_x(\overline{y})\downarrow \\ \uparrow & \text{altrimenti} \end{cases}$$

Questa funzione parziale ricorsiva restituisce il calcolo del programma x con in *input* il vettore $\overline{y}$, se e solo se la funzione calcolata dal programma x, con in *input* il vettore $\overline{y}$, converge. Altrimenti non restituisce nulla: diverge.

In pratica la funzione INT prende in *input* un programma x e un vettore $\overline{y}$, di cui x diventa il programma da eseguire e $\overline{y}$ viene passato come unico *input* per il programma. Dopodiché il comportamento della funzione INT è identico al comportamento del programma con quell'*input*:

- Se il programma x con in *input* $\overline{y}$ termina, allora l'interprete restituisce il risultato calcolato da quel programma x.
- Se invece non termina, allora si comporta in egual maniera e non termina.

Osservazione

L'universalità comporta che si abbia una sola macchina per fare computare qualunque altra MdT.

Esempio

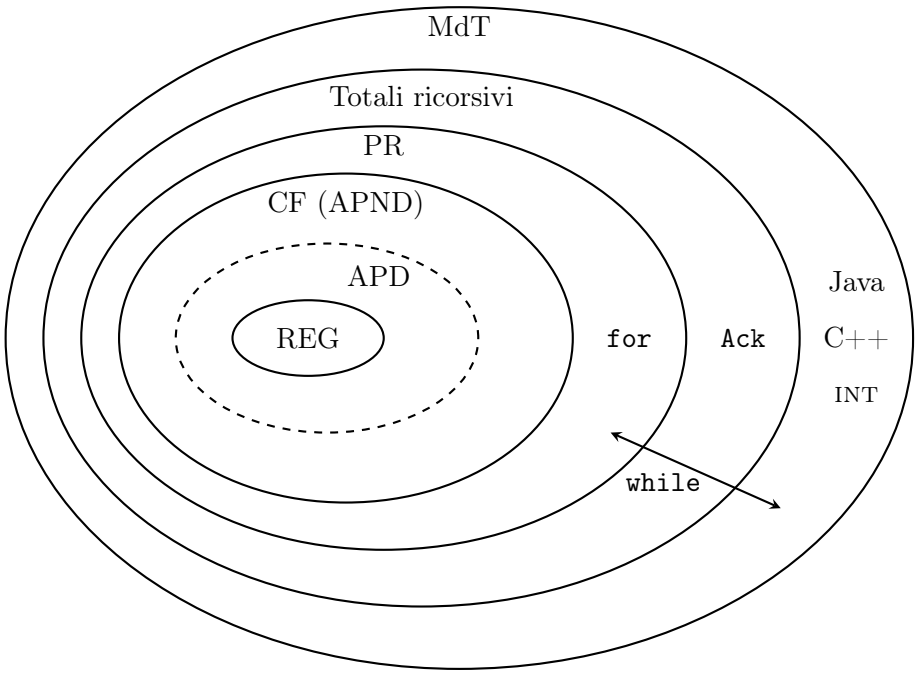
Si costruisce nel dettaglio un interprete INT:

```
input(x)
input(ȳ)
P = fatt(x)
if P == True {
  P(ȳ)
} else {
  while True {
    x = x
  }
}
```

Questo è un interprete nel quale se il programma x è ben costruito, allora il comportamento che l'interprete acquisisce è analogo a quello dell'esecuzione del programma x con in *input* $\overline{y}$.

Si deriva l'equazione fondamentale per cui: l'esecuzione di un interprete sul *Gödel number* di un programma P, con in *input* $\overline{y}$, è uguale a eseguire la funzione calcolata da P con in *input* $\overline{y}$:

$$\varphi_{\text{INT}}(\texttt{gn}(P), \overline{y}) = \varphi_P(\overline{y})$$
$$\downarrow$$
$$\varphi_{\text{INT}}(x, \overline{y}) = \varphi_x(\overline{y})$$



4.3 Specializzatore

Prima di analizzare il concetto di **specializzatore**, è necessario porre prima il seguente teorema.

Teorema 4.3.4: S-M-N

Per ogni $m, n \geq 1$ esiste una funzione totale ricorsiva tale che $\text{SPEC} : \mathbb{N}^{n+1} \to \mathbb{N}$ e per ogni $x, y_1 \dots y_m$:

$$\lambda z_1 \dots z_n.\varphi_x(y_1 \dots y_m, z_1 \dots z_n) = \lambda z_1 \dots z_n.\varphi_{\text{SPEC}(x,y_1 \dots y_m)}(z_1 \dots z_n)$$

dove x è un programma (avente $m+n$ argomenti), $y_1 \dots y_m$ sono valori fissati e $z_1 \dots z_n$ sono variabili.

Il risultato è quello di specializzare un algoritmo A_1 così da produrre un algoritmo A_2. Di conseguenza l'*output* di SPEC è il codice che calcola la funzione specializzata, cioè A_2.

Dimostrazione

Per semplicità si considera la lunghezza dei parametri $m = n = 1$. Sia $y \in \mathbb{N}$, allora fissando y bisogna dimostrare questa uguaglianza:

$$\lambda z.\varphi_x(y, z) = \lambda z.\varphi_{\text{SPEC}(x,y)}(z)$$

La funzione SPEC è un programma che mappa la coppia (x, y) in un nuovo codice:

$$(x, y) \xrightarrow{\text{SPEC}} \begin{bmatrix} \texttt{input(z)} \\ \texttt{x(y,z)} \end{bmatrix}$$

dove x è un programma, y è un *input* fissato e z è la variabile.

Esempio

Si considera il programma $A_1(x, y)$ che esegue la funzione $\lambda xy.x^2 + 2$:

```
input(x)
input(y)
z = x * x
z = z + y
ouput(z)
```

allora fissando $x = 5$, la specializzazione SPEC genera questo programma A_2:

```
input(y)
z = 5 * 5
z = z + y
ouput(z)
```

Uno specializzatore magari più efficiente avrebbe sostituito le operazioni già calcolabili, ottenendo per esempio quest'altro programma A_3:

```
input(y)
z = 25 + y
ouput(z)
```

L'importante è che esista una funzione SPEC, la quale prende un programma e lo trasforma sintatticamente in un codice semanticamente equivalente, con però all'interno dei parametri fissati.

Esempio

Si considera il seguente programma x:

```
input(y)
input(z)
while z <= y { y++ }
ouput(z)
```

Si effettua la specializzazione $\texttt{spec}(x, 5)$, dove 5 è la y fissata:

```
input(z)
y = 5
while z <= y { y++ }
ouput(z)
```

Quindi questo nuovo algoritmo è una specializzazione del precedente x attraverso la funzione $\text{SPEC}(x, 5)$.

Esistono infinite possibilità di specializzazione per un algoritmo, in genere che puntano a migliorare le prestazioni.

Le caratteristiche dei linguaggi Turing completi sono:

- L'esistenza di $\text{INT} \in \mathscr{L}$:

$$\varphi_{\text{INT}}(x, \overline{y}) = \varphi_x(\overline{y})$$

- L'esistenza di SPEC $\in \mathscr{L}$:

$$\lambda\overline{z}.\varphi_{\text{SPEC}(x,\overline{y})} = \lambda\overline{z}.\varphi_x(\overline{y},\overline{z})$$

Tutti i sistemi Turing completi sono caratterizzati dall'avere al loro interno questi due programmi e tutte le loro varianti.

4.3.1 Proiezioni di Futamura (1971)

Queste **proiezioni** consentono di definire con un'equazione il concetto di **compilatore**. Esistono tre proiezioni che hanno il compito di operare sulla sintassi dei programmi:

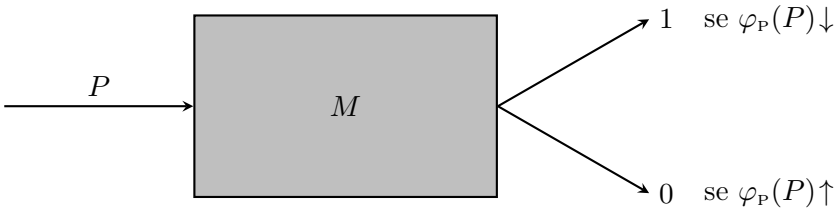
1. Sia un programma $P \in \mathscr{L}_1$, dove $\mathscr{L}_1$ è il linguaggio Java, e sia un interprete INT $\in \mathscr{L}_2$, dove $\mathscr{L}_2$ è il linguaggio C.
   Per ogni $x \in \mathbb{N}$, allora stando alla definizione di interprete l'esecuzione del programma P, con in *input* x, è uguale all'esecuzione dell'interprete INT con in *input* la coppia (P, x):
   $$\varphi_P(x) = \varphi_{\text{INT}}(P, x)$$
   Se si specializza l'interprete rispetto il programma P fissato, allora si ottiene l'uguaglianza con la funzione di SPEC(INT, P) con in *input* x:
   $$\varphi_{\text{SPEC}(\text{INT},P)}(x)$$
   Quindi si passa da $P \in \mathscr{L}_1$ a un programma SPEC(INT, P) $\in \mathscr{L}_2$.

2. Se si esegue la funzione SPEC, con in *input* (INT, P), ovvero il risultato della compilazione, e lo si specializza rispetto al primo argomento, allora si ottiene una funzione di SPEC(SPEC, INT) con in *input* P:
   $$\varphi_{\text{SPEC}}(\text{INT}, P) = \varphi_{\text{SPEC}(\text{SPEC},\text{INT})}(P)$$
   Dunque calcolare la funzione specializzata, ossia specializzare l'interprete con un programma, equivale a specializzare uno specializzatore con un interprete ed eseguire il risultato sul programma P. Ecco che la specializzazione dello specializzatore con un interprete risulta essere il compilatore COMP:
   $$\varphi_{\text{SPEC}(\text{SPEC},\text{INT})}(P) = \varphi_{\text{COMP}}(P)$$

3. L'esecuzione di COMP è φ_{SPEC}(SPEC, INT), di conseguenza si ottiene un generatore di compilatori COMPGEN specializzando la specializzazione di specializzazione e prendendo in *input* l'interprete:
   $$\begin{aligned}\text{COMP} = \varphi_{\text{SPEC}}(\text{SPEC}, \text{INT}) &= \varphi_{\text{SPEC}(\text{SPEC},\text{SPEC})}(\text{INT}) \\ &= \varphi_{\text{COMPGEN}}(\text{INT})\end{aligned}$$

4.4 Classe di funzioni senza un algoritmo

▷ Esiste un algoritmo M tale che: se si dà in *input* un programma P, allora restituisce 1 se $\varphi_{\text{P}}(P)$ termina, altrimenti restituisce 0 se $\varphi_{\text{P}}(P)$ non termina?



La risposta è: non esiste alcun algoritmo per questo problema.

Teorema 4.4.5: Problema della fermata

Data una funzione M con *input* x:

$$M(x) = \begin{cases} 1 & \text{se } \varphi_x(x)\downarrow \\ 0 & \text{se } \varphi_x(x)\uparrow \end{cases}$$

è totale, ma non calcolabile (non ricorsiva).

Dimostrazione

Si suppone per ipotesi assurda che $\tilde{M}$ sia totale ricorsiva:

$$M(x) = \varphi_{\tilde{M}}(x)$$

A questo punto si definisce una funzione H, con in *input* x, nel seguente modo:

$$H(x) = \begin{cases} \uparrow & \text{se } M(x) = 1 \Leftrightarrow \varphi_{\tilde{M}}(x) = 1 \Leftrightarrow \varphi_x(x)\downarrow \\ 0 & \text{se } M(x) = 0 \Leftrightarrow \varphi_{\tilde{M}}(x) = 0 \Leftrightarrow \varphi_x(x)\uparrow \end{cases}$$

Si codifica la funzione H (chiamato programma $\tilde{H}$):

```
input(x)
if φ_INT(M̃, x) == 0 {
  output(0)
} else {
  while true {
    x = x
  }
}
```

- si prende in *input* il codice del programma x;
- si esegue l'interprete INT che esiste nel linguaggio di programmazione impiegato per $\tilde{M}$. Se la sua esecuzione su x restituisce 0, allora si mostra in *output* 0, altrimenti si attiva un ciclo infinito.

Se si manda in esecuzione il programma $\tilde{H}$, allora questo coincide con la funzione H:

$$\varphi_{\tilde{H}}(x) = H(x)$$

$\triangleright$ Cosa succede se $x = \tilde{H}$?
Si ottiene che $\varphi_{\tilde{H}}(x) = H(x)\uparrow$ se e solo se $\varphi_{\tilde{H}}(x)\downarrow$, ma questo è un assurdo:

$$\varphi_{\tilde{H}}(x)\uparrow \iff \varphi_{\tilde{H}}(x)\downarrow$$

Di conseguenza, poiché tutta la costruzione dipende dal fatto che $\tilde{M}$ sia totale ricorsiva, allora non può esistere tale funzione.

Questo risultato determina il seguente corollario.

Corollario 4.4.2

La funzione $H(x, y)$, dove x è un programma e y è un dato, è definita come:

$$H(x,y) = \begin{cases} 1 & \text{se } \varphi_x(y)\downarrow \\ 0 & \text{se } \varphi_x(y)\uparrow \end{cases}$$

Questa funzione è totale, ma non calcolabile (non ricorsiva).

Dimostrazione

Si suppone che $\tilde{H} \in$ Programmi e $H(x, y) = \varphi_{\tilde{H}}(x, y)$, per ogni x, y. Con questa supposizione si prende il caso in cui $x = y$;

$$\lambda x.H(x,x) = \lambda x.\varphi_{\tilde{H}}(x,x)$$

Di conseguenza:

$$\varphi_{\tilde{H}}(x,x) = \begin{cases} 1 & \text{se } \varphi_x(x)\downarrow \\ 0 & \text{se } \varphi_x(x)\uparrow \end{cases}$$

tuttavia questo è assurdo, perché è il medesimo caso della dimostrazione precedente.

Il fatto che non esista il caso particolare del programma - decidibilità dei programmi - su sé stesso, comporta che il problema evidenziato si manifesti in ogni programma.

Teorema 4.4.6

Presa la seguente funzione:

$$T(x) = \begin{cases} 1 & \text{se } \varphi_x \text{ è totale} \\ 0 & \text{se } \varphi_x \text{ non è totale} \end{cases}$$

essa non è calcolabile.

Si ricorda che per definizione una funzione è totale quando per ogni *input* termina:

$$\varphi_x \text{ totale} \overset{\Delta}{\iff} \forall y.\varphi_x(y)\downarrow$$

Dimostrazione

Si suppone che T sia una funzione calcolabile, con $\tilde{T}$ il suo programma.
Si definisce induttivamente una nuova funzione G tale per cui:

$$\begin{cases} G(0) = \mu y.T(y) = 1 \\ G(n+1) = \mu y.(y > G(n) \wedge T(y) = 1) \end{cases}$$

$G(n)$ è il numero di Gödel (n-esima) funzione totale.
Si definisce $H(x)$ come l'x-esima funzione totale, passando x come parametro e sommando 1:

$$H(x) = \varphi_{G(x)}(x) + 1$$

il risultato di H è totale, perciò esiste un numero $n_H \in \mathbb{N}$ che prima o poi verrà individuato dalla funzione G; cioè esiste $\varphi_{G(n_H)} = H$.
Tuttavia se si prende H e le si passa in *input* n_H, allora corrisponde a $\varphi_{G(n_H)}(n_H)$, ma per definizione è anche uguale a $\varphi_{G(n_H)}(n_H) + 1$:

$$H(n_H) = \varphi_{G(n_H)}(n_H) = \varphi_{G(n_H)}(n_H) + 1$$

Da ciò si deriva un assurdo, ossia:

$$\varphi_{G(n_H)}(n_H) = \varphi_{G(n_H)}(n_H) + 1$$

Questa dimostrazione implica che non è possibile ricavare un algoritmo in grado di rispondere con 1 oppure 0 alla domanda φ_x è totale oppure no.

4.4.1 Alcuni problemi insolubili

- Decidere se φ_x è costante:

$$C(x) = \begin{cases} 1 & \text{se } \varphi_x \text{ è costante} \\ 0 & \text{se } \varphi_x \text{ non è costante} \end{cases}$$

  Non è possibile decidere algoritmicamente se un programma calcola sempre lo stesso *output*.

- Decidere se $\varphi_x(y) = z$ per $x, y, z \in \mathbb{N}$:

$$O(x, z) = \begin{cases} 1 & \text{se } \exists y.\varphi_x(y) = z \\ 0 & \text{altrimenti} \end{cases}$$

  Non è possibile decidere algoritmicamente se un programma calcola per un certo *input* un determinato valore.

- Decidere se $\varphi_x = \varphi_y$ sono equivalenti:

$$E(x, y) = \begin{cases} 1 & \text{se } \varphi_x = \varphi_y \\ 0 & \text{se } \varphi_x \neq \varphi_y \end{cases}$$

  Non è possibile decidere algoritmicamente se due programmi calcolano, per un certo *input*, sempre la stessa funzione.

Utilizzare l'informatica per decidere proprietà sui programmi non è possibile.

Linguaggi a struttura di frase

5.1 Equivalenza tra linguaggio e funzione calcolabile

Si definisce un **linguaggio a struttura di frase**, detto anche **linguaggio di Turing**. $\mathscr{L}$ è di Turing se esiste una M MdT tale che $\mathscr{L}$ sia uguale al linguaggio della macchina M, uguale a sua volta all'insieme delle x appartenenti a Σ^* tale per cui la funzione calcolata da M con *input* x è definita:

$$\mathscr{L} = \mathscr{L}(M) = \{x \in \Sigma^* \mid \varphi_M(x)\downarrow\}$$

Un linguaggio di Turing sono tutte le sequenze di *input* tali che la MdT termini in uno stato finale.

Osservazione

Se $\varphi_M(x)$ non è definita (diverge), allora x non appartiene al linguaggio della macchina M:

$$\varphi_M(x)\uparrow \Rightarrow x \notin \mathscr{L}(M)$$

Linguaggi come:

- $\mathscr{L} = \{a^n b^n \mid n \geq 0\}$, appartiene ai CF;
- $\mathscr{L}' = \{a^n b^n c^n \mid n \geq 0\}$, appartiene ai ¬CF;

sono linguaggi di Turing. Infatti entrambi i linguaggi sono **isomorfi**[1] e le singole sottostringhe di a, b e c vengono sfruttate per rappresentare una componente di un vettore:

$$\mathscr{L} \cong \{(x_1, x_2) \mid \underbrace{x_1 = x_2 \geq 0}_{\text{vincolo su } \mathbb{N}^2}\}$$

$$\mathscr{L}' \cong \{(x_1, x_2, x_3) \mid \underbrace{x_1 = x_2 = x_3 \geq 0}_{\text{vincolo su } \mathbb{N}^2}\}$$

[1] Esiste una relazione biunivoca, la quale permette di passare dalla rappresentazione del linguaggio (come $a^n b^n$) a una rappresentazione di vettori di numeri.

Se π è un sottoinsieme di $\mathbb{N}^k$, allora si definisce la funzione f_π prendendo in *input* un vettore, la quale restituisce 1 se e solo se il vettore appartiene a π:

$$\pi \subseteq \mathbb{N}^k, \quad f_\pi(\overline{x}) = 1 \iff \overline{x} \in \pi$$

Allora $\mathscr{L}'$ è definito come:

$$\mathscr{L}' = \{(x_1, x_2, x_3) \mid f_{\pi'}(x_1, x_2, x_3) = 1\}$$
$$\mathscr{L}' \cong \pi'$$

dove $\pi' = \{(x_1, x_2, x_3 \mid x_1 = x_2 = x_3 \geq 0)\}$ e $f_{\pi'}$ è detta **funzione caratteristica**.

Definizione 5.1.1: Funzione caratteristica

Dato un insieme $\pi \subseteq \mathbb{N}^k$, la funzione caratteristica f_π corrisponde a:

$$f_\pi(\overline{x}) = \begin{cases} 1 & \text{se } \overline{x} \in \pi \\ 0 & \text{se } \overline{x} \notin \pi \end{cases}$$

5.1.1 Struttura dell'insieme L_0

Dominio di una funzione parziale ricorsiva

Il dominio di una funzione parziale è l'insieme delle x tale che la funzione parziale è definita:

$$\texttt{dom}(\varphi) = \{x \mid \varphi(x)\downarrow\}$$

Un linguaggio $\mathscr{L}$ è di Turing se esiste una x tale per cui $\mathscr{L}$ è uguale al dominio della funzione parziale ricorsiva su x:

$$\exists x : \mathscr{L} = \texttt{dom}(\varphi_x) \Rightarrow \mathscr{L} \in L_0$$

5.2 Insiemi ricorsivi e ricorsivamente enumerabili

Se φ_x è parziale ricorsiva, allora il dominio di φ_x viene rappresentato con la notazione W_x:

$$\texttt{dom}(\varphi_x) \doteq W_x$$

Definizione 5.2.2: Insiemi ricorsivamente enumerabili (RE)

Un insieme I, sottoinsieme dei numeri naturali, è ricorsivamente enumerabile (RE) se esiste un programma P tale che I sia uguale al dominio di φ_P corrispondente a W_P:

$$I \subseteq \mathbb{N} : I \in \text{RE} \Rightarrow \exists P : I = \texttt{dom}(\varphi_P) = W_P$$

Definizione 5.2.3: Insiemi ricorsivi (REC)

Un insieme I, sottoinsieme dei numeri naturali, è ricorsivo (REC) se esiste un programma P tale che $\varphi_P(x)$ sia uguale a 1 se $x \in I$, altrimenti uguale a 0:

$$I \subseteq \mathbb{N} : I \in \text{REC} \Rightarrow \exists P : \varphi_P(x) = \begin{cases} 1 & \text{se } x \in I \\ 0 & \text{se } x \notin I \end{cases}$$

dove φ_P è totale ricorsiva.

La differenza che contraddistingue questi due insiemi appena definiti è che:

- gli insiemi ricorsivamente enumerabili RE hanno il dominio della **funzione parziale ricorsiva** calcolata da un generico programma P;
- gli insiemi ricorsivi REC hanno un programma P che calcola la **funzione caratteristica** di I.

▷ In che relazione stanno le famiglie di funzioni REC e RE?
Se $I \in \text{RE}$, allora $I = W_x$, cioè è l'insieme delle x tale che $\varphi_P(x)$ è definita:

$$I \in \text{RE} \Rightarrow I = W_P = \{x \mid \varphi_P(x)\downarrow\}$$

Si può costruire un programma Q tale per cui $\varphi_Q(x)$ sia uguale a 1 se $x \in I$, altrimenti diverge:

$$\exists Q : \varphi_Q(x) = \begin{cases} 1 & x \in I \\ \uparrow & \text{altrimenti} \end{cases}$$

Questa funzione prende il nome di **funzione semi-caratteristica** di I.

```
input(x)
if φ_INT(P,x)↓ {
  output(1)
}
```

Questo programma riesce ad affermare in un tempo finito se un programma P generico termina.

Non esiste una condizione `else` per questo programma, perché altrimenti si affermerebbe che in un tempo finito un programma P generico non termina.

Ogni insieme ricorsivamente enumerabile induce una funzione semi-caratteristica programmabile, mentre ogni insieme ricorsivo induce una funzione caratteristica totale programmabile.

$$\text{RE} : \text{semi-caratteristica} = \text{REC} : \text{totale-caratteristica}$$

Si suppone che $I \in \text{REC}$, allora per definizione implica che:

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

Osservazione

Per i linguaggi ricorsivi è possibile affermare sia 1 sia 0. Di conseguenza è possibile affermare anche solo 1, "silenziando" lo 0.

Quindi si riesce a passare da una funzione caratteristica totale a una funzione semi-caratteristica. Esiste un programma Q con il seguente codice:

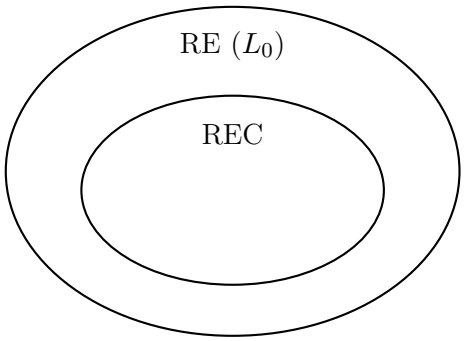
```
input(x)
if φ_INT(P,x) == 1 {
  output(1)
} else {
  while true {
    x = x
  }
}
```

$$\varphi_Q(x) = \begin{cases} 1 & x \in I \\ \uparrow & \text{altrimenti} \end{cases}$$

In particolare l'insieme di x tale che $\varphi_Q(x)$ sia definito è uguale all'insieme di x tale per cui $x \in I$, ovvero uguale all'insieme I:

$$\{x \mid \varphi_Q(x)\downarrow\} = \{x \mid x \in I\} = I$$

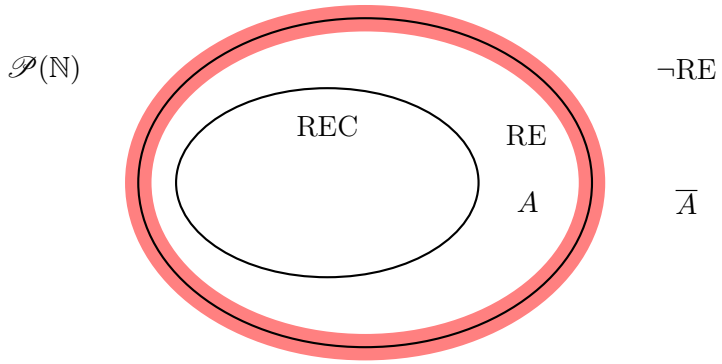
Ciò implica che I è RE. Ne consegue che $\text{REC} \subseteq RE$:



Esempio

Insiemi contenuti in REC possono essere: $\emptyset$, $\mathbb{N}$, $2\mathbb{N}$, $2\mathbb{N}+1$, $\{S \mid |S| < \omega\}$ (insieme finito), REG, CF, ecc.

▷ Esiste qualcosa che distingue RE da REC, oppure trattano la stessa cosa e tutto collassa in RE?
L'idea consiste nel dimostrare che se dovesse esistere in RE (al di fuori di REC) un insieme A, allora al di fuori di RE dovrebbe esistere il suo complementare $\overline{A}$. Pertanto $\overline{A}$ non è Turing calcolabile.



Tutto ciò che sta al di fuori dell'insieme RE non è calcolabile.

Teorema 5.2.1: Emil Post (1944)

Sia $A \subseteq \mathbb{N}$, cioè $A \in \mathscr{P}(\mathbb{N}) = 2^{\mathbb{N}}$, è ricorsivo se e solo se A e il suo complemento $\overline{A} \doteq \mathbb{N} \setminus A$ sono ricorsivamente enumerabili:

$$A \subseteq \mathbb{N} : A \in \text{REC} \iff A \in \text{RE} \wedge \overline{A} \in \text{RE}$$

Dimostrazione

Si dimostrano entrambe le implicazioni:

- **Prima implicazione $\Rightarrow$**:
  - Sia $A \in$ REC, allora esiste un programma $\tilde{A}$ tale che:

$$\varphi_{\tilde{A}} = \begin{cases} 1 & \text{se } x \in A \\ 0 & \text{se } x \notin A \end{cases}$$

Poiché REC $\subseteq$ RE, allora A $\subseteq$ RE.

– Per verificare $\overline{A} \in$ RE, si costruisce il seguente programma Q:

```
input(x)
if φ_INT(Ã,x) == 0 {
  output(1)
} else {
  while true {
    x = x
  }
}
```

Si prende come *input* l'insieme di x tale che $\varphi_Q(x)$ converga, il quale corrisponde all'insieme delle x tale per cui $\varphi_{\text{INT}}(\tilde{A}, x) = 0$, cioè uguale all'insieme delle x tale che $\varphi_{\tilde{A}}(x) = 0$:

$$\{x \mid \varphi_Q(x)\downarrow\} = \{x \mid \varphi_{\text{INT}}(\tilde{A}, x) = 0\} = \{x \mid \varphi_{\tilde{A}}(x) = 0\}$$

Ma $\varphi_{\tilde{A}} = 0$ se e solo se x non appartiene ad A, ossia appartiene al suo complementare:

$$\{x \mid \varphi_{\tilde{A}}(x) = 0\} = \{x \mid x \notin A\} = \overline{A}$$

Ne consegue che $\overline{A}$ è corrispondente al dominio di Q e perciò appartiene a RE:

$$\overline{A} = W_Q \Rightarrow \overline{A} \in \text{RE}$$

- **Seconda implicazione** $\Leftarrow$: si suppongono sia $A \in$ RE sia $\overline{A} \in$ RE. Ciò vuole dire che si può costruire la funzione semi-caratteristica:

$$\varphi_{\tilde{A}}(x) = \begin{cases} 1 & \text{se } x \in A \\ \uparrow & \text{altrimenti} \end{cases}$$

$$\varphi_{\tilde{\overline{A}}}(x) = \begin{cases} 1 & \text{se } x \notin A \\ \uparrow & \text{altrimenti} \end{cases}$$

Sicuramente $x \in A$ oppure $x \notin A$, perciò mandando in esecuzione entrambe le funzioni una delle due deve terminare; la prima funzione che termina dichiara dove la x appartenga.

```
input(x)
if φ_Ã(x) == 1 then output(1) ||
    if φ_~Ã(x) == 1 then output(0)
```

Questo programma Q ha una funzione φ_Q totale ricorsiva:

$$\varphi_Q = \begin{cases} 1 & \text{se } x \in A \\ 0 & \text{se } x \notin A \end{cases}$$

Ciò implica che l'insieme A sia ricorsivo.

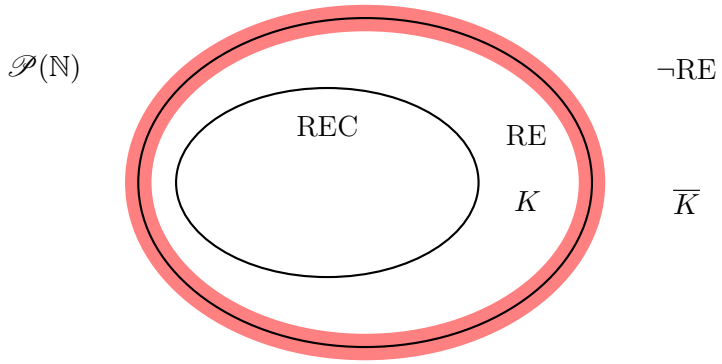
Definizione 5.2.4: Insieme K

K è l'insieme dei programmi x che terminano su loro stessi. Quindi K è l'insieme x tale per cui x appartiene al dominio di x:

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

Teorema 5.2.2

K è ricorsivamente enumerabile, ma non ricorsivo.



Per dimostrare che $K \in$ RE, occorre verificare che il dominio sia una funzione parziale ricorsiva.

Dimostrazione

Si suppone che $K \in$ RE e si fornisce il seguente programma $\tilde{K}$:

```
1 input(x)
2 if φ_x(x)↓ == 1 then output(1)
```

$$\varphi_{\tilde{K}}(x) = \begin{cases} 1 & \text{se } x \in K \Leftrightarrow \varphi_x(x)\downarrow \\ \uparrow & \text{altrimenti} \end{cases}$$

Ciò implica che $K = W_{\tilde{K}}$, ovvero $K \in \text{RE}$.
Si suppone che K sia ricorsivo. Per il teorema di Post, $\overline{K} \in \text{RE}$, allora esiste un programma P tale che $\overline{K}$ sia uguale al dominio di P, cioè all'insieme di x tale che $\varphi_P(x)$:

$$\overline{K} \in \text{RE} \Rightarrow \exists P : \overline{K} = W_P = \{x \mid \varphi_P(x)\downarrow\}$$

dove $\overline{K} = \{x \mid \varphi_x(x) \uparrow\} \subseteq \mathbb{N}$. Poiché P è un programma, appartiene a $\overline{K}$ se e solo se $P \in W_P$, se e solo se a sua volta $\varphi_P(P)\downarrow$:

$$\begin{aligned} P \in \overline{K} &\iff P \in W_P \\ &\iff \varphi_P(P)\downarrow \end{aligned}$$

Tuttavia $P \in \overline{K}$ anche se e solo se $\varphi_P(P) \uparrow$:

$$P \in \overline{K} \iff \varphi_P(P)\uparrow$$

Allora per transitività il programma P (che si è supposto esistere per assurdo) è tale per cui esso termina se e solo se non termina:

$$\varphi_P(P)\uparrow \iff \varphi_P(P)\downarrow$$

Questo è un assurdo e il programma P non può esistere.

In conclusione, tutto ciò che si è visto fin'ora:

- ogni volta che c'è un insieme che distingue i ricorsivamente enumerabili dai ricorsivi, il complemento di questo insieme sta fuori dai ricorsivamente enumerabili;
- l'insieme K codifica come linguaggio l'insieme delle istruzioni di programmi che se prendono in *input* loro stesse, allora terminano. Il suo negato, l'insieme $\overline{K}$, codifica l'insieme di istruzioni di programmi che se prendono in *input* loro stesse, allora non terminano.

Questi due insiemi devono stare per forza separati: uno appartenente a RE, mentre l'altro appartenente a $\neg$RE.

Dopodiché dal teorema di Post si deriva che se si hanno due insiemi, l'uno il complemento dell'altro ed entrambi ricorsivamente enumerabili, allora si riesce a comporre questi due programmi e costruire una funzione caratteristica, ovvero dimostrare che entrambe sono ricorsive.

5.2.1 Caratterizzazione dell'insieme RE

Teorema 5.2.3: Caratterizzazione di RE (S.Kleene)

Le seguenti affermazioni sono equivalenti:

1. $A \subseteq \mathbb{N}$ è ricorsivamente enumerabile.
2. $A = \texttt{range}(\varphi_P) = \{y \mid \exists x.y = \varphi(x)\}$, con φ_P parziale ricorsiva.
3. $A = \emptyset$ oppure $A = \texttt{range}(f)$, con f totale ricorsiva.

Questi punti significano in modo semplice che:

1. A è un insieme in RE.
2. A è l'*output* di un programma, il quale può non terminare.
3. A è vuoto oppure è l'*output* di un programma che termina sempre.

Dimostrazione

Si dimostrano le tre implicazioni $1 \Rightarrow 2 \Rightarrow 3 \Rightarrow 1$:

- **Prima implicazione** $1 \Rightarrow 2$: per ipotesi sia $A \in$ RE e dunque per definizione $A = W_x$, con x un programma generico.
  Si definisce la funzione parziale ricorsiva $H(y)$ come:

$$H(y) = \begin{cases} y & \text{se } \varphi_x(y)\downarrow \\ \uparrow & \text{altrimenti} \end{cases}$$

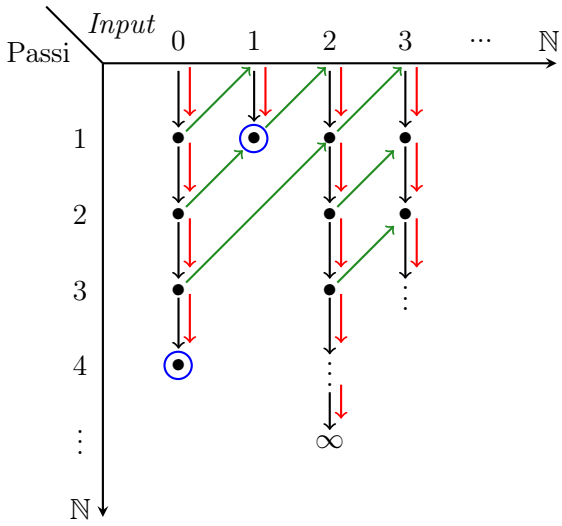
  Si scrive il programma Q tale per cui $\varphi_Q = H$:

```
1 input(y)
2 if φ_INT(x,y)↓ then output(y)
```

  Da questo si deriva che:

$$\begin{aligned} \texttt{range}(\varphi_Q) &= \texttt{range}(H) \\ &= \{y \mid \exists x.H(x) = y\} \\ &= \{y \mid \varphi_x(y)\downarrow\} \\ &= W_x = A \end{aligned}$$

- **Seconda implicazione** $2 \Rightarrow 3$: per ipotesi si sa che $A = \texttt{range}(\varphi_x)$, di conseguenza occorre verificare i casi del punto (3):
  - Se $A = \texttt{range}(\varphi_x)$ e $A = \emptyset$, allora φ_x non termina mai.
  - Se $A = \texttt{range}(\varphi_x)$ e $A \neq \emptyset$, allora si esplora il campo di definizione dell'algoritmo x di φ_x: si dispone sulle colonne tutti gli *input* del programma x, mentre sulle righe il numero di passi effettuati durante l'esecuzione:



Questa tecnica prende il nome di ***dovetail***: dopo aver eseguito un passo i-esimo, con un certo *input* j-esimo, si torna indietro di un passo e si esegue l'*input* successivo, dopodiché si riprende con il passo successivo dell'*input* j-esimo. Nell'atto pratico si riesce a spazzare l'intero campo e quando si trova un punto di terminazione lo si attribuisce alla funzione f. Poiché $A \neq \emptyset$, allora esiste un *input* z tale che $\varphi_x(z) = y_0 \in A$ dopo n_0 passi di *dovetail*:

$$\begin{cases} f(0) = y_0 \\ f(n+1) = \begin{cases} y' & \text{dopo } n_0 + n + 1 \text{ passi: } \varphi_x(z') = y',\ z' \in \mathbb{N} \text{ e } y' \in A \\ f(n) & \text{altrimenti} \end{cases} \end{cases}$$

Quindi la funzione f, dato un certo numero in *input*, calcola il numero di passi necessari del *dovetail* per trovare il valore successivo al primo valore di terminazione. Se non trova un nuovo punto di terminazione z' - un *input* per cui la funzione termina con un y' -, allora si passa il valore precedente.

> Questo è un algoritmo in grado di enumerare tutti i punti in cui un programma, che può non terminare, termina.

Ne consegue che $A = \texttt{range}(f)$, con f totale ricorsiva.

- **Terza implicazione** $3 \Rightarrow 1$: si hanno due ipotesi derivate dal punto (3):
  - Se $A = \texttt{range}(f)$ e $A = \emptyset$, allora $A \in \text{RE}$ perché $A = W_{\texttt{while true \{ x = x \}}}$.
  - Se $A = \texttt{range}(f)$ e $A \neq \emptyset$, con f totale ricorsiva, allora si ha che:

$$H(x) = \mu z.(f(z) - x = 0)$$

    è parziale ricorsiva, perché: $f(z)$ restituisce sempre un risultato (è totale); la sottrazione $f(z) - x$ è una primitiva ricorsiva; infine il *test* 0 permette la minimizzazione μ.

  Quindi esiste un programma P tale per cui $\varphi_P = H$, inoltre A è il *range* di f: tutte le volte che la sottrazione $f(z) - x$ risulta 0, cioè x è uguale all'*output* di $f(z)$, allora x appartiene ad A, che è uguale al *range* di f:

$$\begin{aligned} A &= \{x \mid H(x)\downarrow\} \\ &= \{x \mid \varphi_P(x)\downarrow\} \\ &= W_P \in \text{RE} \end{aligned}$$

Affermare che un insieme A è ricorsivamente enumerabile (RE) equivale a dire che è uguale all'insieme di *output* generati da un programma, dove questo insieme è vuoto (il programma non termina) oppure è l'*output* di un programma che termina sempre.

5.2.2 Proprietà degli insiemi RE e REC

Teorema 5.2.4: Prima caratterizzazione di REC

Le seguenti affermazioni sono equivalenti:

- A è ricorsivo (REC).
- $A = \emptyset$ oppure $A = \texttt{range}(f)$, con f totale ricorsiva non decrescente.

Dimostrazione

Si dimostrano entrambe le implicazioni dell'equivalenza:

- **Prima implicazione** $\Rightarrow$: l'ipotesi di partenza è che A sia ricorsivo.

- Se $A = \emptyset$, allora è ovvio, poiché l'insieme vuoto è ricorsivo.
- Se $A = \texttt{range}(f)$, poiché $A \subseteq \mathbb{N}$, allora $a = \texttt{min}(A)$ e per ricavarlo si utilizza la seguente funzione:

$$\begin{cases} f(0) = a \\ f(n+1) = \begin{cases} n+1 & n+1 \in A \\ f(n) & n+1 \notin A \end{cases} \end{cases}$$

Questa funzione è totale ricorsiva dato che A è un insieme ricorsivo: è possibile decidere con un `if/else` l'appartenenza o no di un punto. Inoltre $A = \texttt{range}(f)$, ne consegue che la funzione sia non decrescente.

- **Seconda implicazione** $\Leftarrow$: per ipotesi $A = \emptyset$ oppure $A = \texttt{range}(f)$, con f totale ricorsiva non decrescente.
  Allora nel primo caso, A è ovviamente ricorsiva. Nel secondo caso bisogna costruire una funzione in grado di determinare se $x \in A$ oppure $x \notin A$:
  - Se A è finito, allora A è ricorsivo.
  - Se $|A| = \omega$, allora $f(x)$ risulta essere:

$$f(x) = \begin{cases} 1 & \text{se } x \in \{f(0), \ldots, f(z_x)\}, \text{ dove } z_x = \mu y.f(y) > x \\ 0 & \text{altrimenti} \end{cases}$$

L'algoritmo prende x e cerca il primo punto di A che lo supera. Una volta trovato non serve verificare tutti i punti successivi a quel punto, perché essendo una funzione non decrescente, se $x \in A$, allora sta per forza nei punti tra $f(0)$ e f che supera x.

Teorema 5.2.5: Seconda caratterizzazione di REC

Le seguenti affermazioni sono equivalenti:

- A è ricorsivo (REC) e $|A| = \omega$.
- $A = \texttt{range}(f)$, con f totale ricorsiva crescente.

Dimostrazione

Si dimostrano entrambe le implicazioni dell'equivalenza:

- **Prima implicazione** $\Rightarrow$: per ipotesi A è ricorsivo e $|A| = \omega$. Data questa

ipotesi si definisce:

$$\begin{cases} f(0) = \mu n.n \in A \\ f(n+1) = \mu m.(m > f(n) \wedge m \in A) \end{cases}$$

La condizione per la quale $|A| = \omega$ permette di trovare sempre un valore maggiore. Ne consegue che f è crescente e totale ricorsiva.

> Infatti si cadrebbe nell'assurdo se $\mu m.(m > f(n) \wedge m \in A)\uparrow$.
> Ne conseguirebbe che $\forall m > f(n) : m \notin A \iff A \subseteq [0, f(n)]$.

- **Seconda implicazione $\Leftarrow$**: si suppone che A sia il $\texttt{range}(f)$, con f una funzione totale ricorsiva crescente. Quindi $A = \{f(0), f(1), \dots, f(n), \dots\}$, dove $\forall n > m$ si ha che $f(n) > f(m)$ e $|A| = \omega$.
  Se $x \in \mathbb{N}$, allora esiste $m \in \mathbb{N}$ tale che $x \leq f(m)$. Si definisce la funzione caratteristica di A come:

$$f_A(A) = \begin{cases} 1 & \text{se } x \in \{f(0), \dots, f(m_x)\} \\ 0 & \text{altrimenti} \end{cases}$$

  Ne consegue che A sia ricorsivo.

Teorema 5.2.6

Se $A \in$ RE e $|A| = \omega$, allora esiste $B \in$ REC, tale che $B \subseteq A$ e $|B| = \omega$.

Dimostrazione

Sia $A \in$ RE e $|A| = \omega$, allora per il teorema di caratterizzazione di Kleene esiste una funzione f totale ricorsiva, tale per cui $A = \texttt{range}(f)$. Si definisce una nuova funzione:

$$\begin{cases} g(0) = f(0) \\ g(n+1) = f(\mu y.f(y) > g(n)) \end{cases}$$

Ne consegue che la funzione sia crescente, perché si cerca il minimo valore y maggiore stretto di quello precedente. Inoltre g è totale, poiché $|A| = \omega$, ed è ricorsiva per definizione. Pertanto $B = \texttt{range}(g)$ e ne consegue che $B \subseteq A$ è ricorsiva e $|B| = \omega$.

5.3 Teoremi di ricorsione e di Rice

5.3.1 Primo teorema di ricorsione

Teorema 5.3.7: Punto fisso (S.Kleene, 1938)

Sia t una funzione totale ricorsiva. Allora esiste un programma $\tilde{e}$ tale che la funzione calcolata da $\tilde{e}$ sia uguale alla funzione calcolata da $t(\tilde{e})$:

$$t \text{ totale ricorsiva} \Rightarrow \exists \tilde{e} : \varphi_{\tilde{e}} = \varphi_{t(\tilde{e})}$$

Dimostrazione

Sia u un programma generico. Allora si definisce la seguente funzione di un altro programma Q:

$$\varphi_Q(u,x) = \begin{cases} \varphi_{\varphi_u(u)}(x) & \text{se } \varphi_u(u)\downarrow \\ \uparrow & \text{altrimenti} \end{cases}$$

Il programma Q è descritto come:

```
input(u)
input(x)
if φ_INT(u,u) == z then φ_INT(z,x)
```

Con SMN, se si fissa u, allora $\varphi_Q(u,x) = \varphi_{g(Q,u)}(x)$ dove g è totale ricorsiva; ovvero $\lambda u.g(Q,u)$ è totale ricorsiva.
Per ipotesi del teorema, t è totale ricorsiva e di conseguenza se si compongono due funzioni totali ricorsive, allora si ottiene ancora una funzione totale ricorsiva: $t \circ g$ è totale ricorsiva. Questo implica l'esistenza di una particolare MdT v, tale per cui la funzione calcolata da quel programma è uguale alla funzione calcolata componendo t e g:

$$\Rightarrow \exists v(\text{MdT}) : \varphi_v(u) = t(g(Q,u))$$

Essendo u un programma generico (preso all'inizio della dimostrazione), si sostituisce u con v, il quale è una particolare MdT, così da ottenere:

$$\varphi_{g(Q,v)}(x) = \begin{cases} \varphi_{\varphi_v(v)}(x) & \text{se } \varphi_v(v)\downarrow \\ \uparrow & \text{altrimenti} \end{cases}$$

Tuttavia non potrà mai avvenire che $\varphi_{g(Q,v)}(x)\uparrow$, perché componendo due funzioni totali la condizione $\varphi_v(v)\downarrow$ risulta essere sempre vera. Ciò implica che:

$$\varphi_{g(Q,v)}(x) = \varphi_{\varphi_v(v)}(x) = \varphi_{t(g(Q,v))}(x)$$

Ponendo $\tilde{e} \doteq g(Q,v)$, si ottiene che:

$$\varphi_{\tilde{e}}(x) = \varphi_{t(\tilde{e})}(x)$$

Quindi per ogni funzione totale, esiste sempre un **punto fisso**: un programma tale per cui la funzione calcolata da questo programma ($\varphi_{\tilde{e}}$) sia uguale alla funzione calcolata dal programma trasformato da t ($\varphi_{t(\tilde{e})}$), dove t è una qualsiasi funzione totale ricorsiva.

5.3.2 Secondo teorema di ricorsione

Teorema 5.3.8: Punto fisso

Sia $f : \mathbb{N}^2 \to \mathbb{N}$ una funzione totale ricorsiva. Allora esiste una funzione totale ricorsiva g, tale che per ogni y la funzione calcolata secondo f di $g(y)$ e y sia uguale alla funzione calcolata in $g(y)$:

$$\Rightarrow \exists g : \forall y. \varphi_{f(g(y),y)} = \varphi_{g(y)}$$

Dimostrazione

Si definisce una funzione di tre argomenti:

$$\varphi_?(x, y, z) = \begin{cases} \varphi_{f(\varphi_x(x),y)}(z) & \text{se } \varphi_x(x)\downarrow \\ \uparrow & \text{altrimenti} \end{cases}$$

Si costruisce questo programma ancora ignoto (?):

```
input(x)
input(y)
input(z)
if φ_INT(x,x) == m then φ_INT(f(m,y),z)
```

Per SMN, fissando i parametri x e y e rimpiazzando S al "?", si ottiene:

$$\varphi_?(x, y, z) = \varphi_{S(x,y)}(z)$$

dove S è totale ricorsiva. Quindi quanto ricavato corrisponde ulteriormente a:

$$\varphi_{S(x,y)}(z) = \varphi_{f(\varphi_x(x),y)}(z), \quad \text{se } \varphi_x(x)\downarrow$$

Essendo S totale ricorsiva, allora è calcolata da un programma. Infatti esiste un programma Q MdT tale che la funzione S, avente come parametri x e y, è uguale alla funzione parziale calcolata su Q dei medesimi parametri:

$$\Rightarrow \exists Q(\text{MdT}) : S(x, y) = \varphi_Q(x, y)$$

Per SMN, fissando il parametro y, si ottiene:

$$S(x,y) = \varphi_Q(x,y) = \varphi_{t(Q,y)}(x)$$
$$\varphi_{t(Q,y)}(x) = \varphi_{t(y)}(x), \quad \text{se } Q \text{ è costante}$$

Si definisce la funzione $g(y) \doteq \varphi_{t(y)}(t(y)) = S(t(y), y)$, che è totale ricorsiva. Si ottengono le seguenti uguaglianze:

$$\begin{aligned}
\varphi_{g(y)} &= \varphi_{\varphi_{t(y)}(t(y))} \\
&\doteq \varphi_{S(t(y),y)} \\
&= \varphi_{f(\varphi_{t(y)}(t(y)),y)}, \quad \text{perché } \varphi_{t(y)}(t(y))\downarrow \\
&= \varphi_{f(g(y),y)}
\end{aligned}$$

Tramite questa cascata di equazioni, è sempre possibile costruire la funzione $g(y)$ come **punto fisso**: trasforma qualunque y in un valore, che è punto fisso dell'equazione.

Caso particolare

Corollario 5.3.1

Se $f(x,y)$ è totale ricorsiva, allora esiste R MdT tale che:

$$\varphi_R(y) = f(R,y)$$

Significa che: se si dispone di una qualsiasi funzione f totale ricorsiva, avente due parametri, allora esiste un programma R che, se mandato in esecuzione, corrisponde al calcolo della funzione f dove il suo *input* è il suo stesso codice.

Corollario 5.3.2: Quine code

Se $f(x,y) = x$, ovvero la 1^a proiezione (primitiva ricorsiva), allora esiste un programma R tale che, per ogni y, $\varphi_R(y) = f(R,y)$; cioè $\varphi_R(y) = R$.

La conseguenza del secondo teorema di ricorsione è che: esiste un programma che se mandato in esecuzione, allora produce in *output* il suo stesso codice. Questi programmi prendono il nome di ***Quine code***.

Bits di virologia sintetica

Un ***malware*** è un programma (MdT) che agisce in modo malevolo attraverso un **processo infettivo**.

- Si infettano sistemi (programmi).

- L'infezione può essere controllata mediante *trigger*: il *malware* rimane silente finché non avviene l'evento di sblocco.

Vengono forniti gli elementi $v, P \in \text{MdT}$, dove v è un *virus*, P è un sistema da infettare e l'*input trigger* $x \in \text{Dato} = \mathbb{N}$. Allora l'**infezione** è una funzione che prende in *input* il *virus* v, il sistema P e il *trigger* x, e restituisce un **sistema infetto**:

$$f : \mathbb{N}^3 \to \mathbb{N}$$
$$f : \underbrace{\text{MdT}}_{v} \times \underbrace{\text{MdT}}_{S} \times \underbrace{\mathbb{N}}_{x} \to \text{MdT}$$

v è un *virus* se: eseguire v in un programma P, con in *input* un *trigger* x, è uguale ad agire con una funzione f avente come parametri v, P e x:

$$\varphi_v(P, x) = f(v, P, x)$$

Quindi un *virus* esiste nel momento in cui l'esecuzione di un algoritmo corrisponde al calcolare una funzione con lo stesso codice eseguito; 2° teorema di ricorsione.

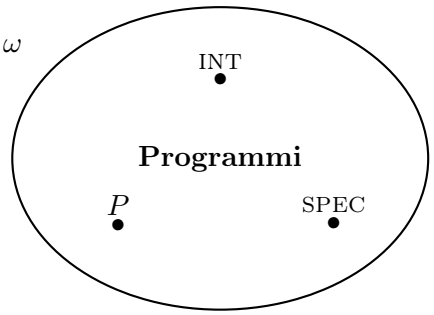
Esempio: *I ♡ you!*

```
f(v,mailbox,Agenda):
  input(v,mailbox,Agenda)
  x = Agenda
  while x != null {
    mailbox = mailbox(header(x),v)
    x = tail(x)
  }
  mailbox = mailbox("joker@dom.com",A)
  send(mailbox)
```

5.3.3 Proprietà dei programmi

L'informatica è la scienza e la tecnologia dell'informazione. Quest'ultima viene monitorata attraverso algoritmi, ossia programmi descritti mediante linguaggi di programmazione. L'universo $\mathbb{U}$ è il mondo dei programmi; un oggetto appartenente a questo insieme è un qualunque programma, cioè un particolare algoritmo.

$$|\mathbb{U}| = |\mathbb{N}| = \omega$$



▷ Cos'è una proprietà di un programma?
Per **proprietà** si intende una descrizione che rappresenta un certo insieme di programmi.

Esempio

Proprietà che definiscono sottoinsiemi di programmi in $\mathbb{U}$ possono essere:

$$\pi = \{P \mid |P| = 50 \text{ locazioni}\}$$
$$\pi = \{P \mid \forall x.\varphi_P(x) = 5\}$$
$$\pi = \{P \mid \texttt{time}(\varphi_P(x)) \leq |x|^2\}$$
$$\pi = \{P \mid \varphi_P = f\}$$

Definizione 5.3.5: Proprietà

Le proprietà dei programmi sono racchiuse in un certo insieme $\pi \subseteq \mathbb{U}$.

Quindi esiste una certa MdT, la quale dato un programma P in *input* risponde "sì" o "no" in *output*, a seconda che appartenga o meno a una certa proprietà π da dover verificare.

Definizione 5.3.6: Proprietà estensionale

$\pi \subseteq \mathbb{U}$ è estensionale se $x \in \pi$ e $\varphi_x = \varphi_y$, allora $y \in \pi$.

Il concetto di **estensionalità** significa che la proprietà cattura ciò che calcola un programma e non le sue caratteristiche di scrittura. Se un programma x soddisfa una certa proprietà π e un altro programma y possiede lo stesso comportamento di x, allora anche y possiede quella stessa proprietà π. Tutto ciò che non è estensionale si dice **intensionale**.

La complessità degli algoritmi non fornisce proprietà estensionali dei programmi.

Teorema 5.3.9: Henry Gordon Rice (1953)

Sia $\pi \subseteq \mathbb{U}$ una certa proprietà estensionale. Allora π è ricorsivo se e solo se $\pi = \emptyset$ oppure $\pi = \mathbb{U}$.

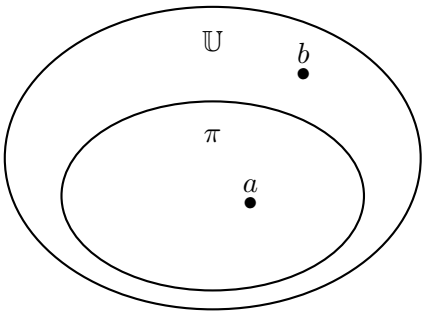
$$\pi \in \text{REC} \iff \pi = \emptyset \vee \pi = \mathbb{U}$$

Le uniche due proprietà estensionali ricorsive sono: essere un programma e non essere un programma. Di conseguenza nessun altra proprietà estensionale è ricorsiva: per nessuna proprietà dei programmi, la quale predica su ciò che i programmi fanno (non su come sono descritti), non può esistere una macchina in grado di affermare "sì" o "no" se la proprietà è vera o meno.

Dimostrazione

Si dimostrano entrambe le implicazioni dell'equivalenza:

- **Prima implicazione** $\Leftarrow$: è banale, poiché se $\pi = \emptyset$ oppure $\pi = \mathbb{U}$ allora sono linguaggi regolari. Ciò implica che siano linguaggi ricorsivi.
- **Seconda implicazione** $\Rightarrow$: si suppone per assurdo che $\pi \neq \emptyset$ e $\pi \neq \mathbb{U}$, con π una proprietà estensionale e ricorsiva.
  Poiché si ipotizza π essere ricorsivo, allora esiste una macchina, che prende in *input* un programma P, in grado di rispondere "sì" se $P \in \pi$ oppure "no" se $P \notin \pi$. Tuttavia aver supposto che $\pi \neq \emptyset$ significa affermare che internamente a $\mathbb{U}$ esiste un sottoinsieme stretto non vuoto; perciò è presente almeno un punto sia interno sia esterno a π.
  Questi due programmi, di cui non importa la loro struttura, vengono chiamati rispettivamente a e b:



Si costruisce la seguente funzione totale ricorsiva:

$$H(x) = \begin{cases} b & \text{se } x \in \pi \\ a & \text{se } x \notin \pi \end{cases}$$

```
input(x)
if φ_{M_π}(x) == 1 then output(b)
                   else output(a)
```

Per il primo teorema di ricorsione di Kleene: $\exists n_0 . \varphi_{n_0} = \varphi_{H(n_0)}$. Poiché $\pi \subset \mathbb{U}$, allora n_0 deve essere un programma e da questo derivano due possibilità:

1. Se $n_0 \in \pi$: $\varphi_{n_0} = \varphi_{H(n_0)}$, per estensionalità, allora $H(n_0) \in \pi$. Si arriva all'assurdo perché $H(n_0)$ risulta essere b, che non appartiene a π.

2. Se $n_0 \notin \pi$: $\varphi_{n_0} = \varphi_{H(n_0)} = \varphi_a$, ciò implica che $a \notin \pi$. Tuttavia è assurdo, perché a è stato scelto interno a π.

Dunque è possibile decidere se un programma soddisfa una proprietà estensionale se e solo se questa sia banale, cioè $\pi = \emptyset$ oppure $\pi = \mathbb{U}$. Invece tutte le altre proprietà estensionali (non banali) non risultano essere decidibili.
Le conseguenze del teorema di Rice sono:

- Sia una funzione f specifica. Allora la proprietà di calcolare la funzione f, cioè tutti i programmi tali che la funzione calcolata dai programmi sia uguale a f, non è un insieme ricorsivo.

$$\pi_f = \{P \mid \varphi_P = f\} \notin \text{REC}$$

  - È estensionale: se $P \in \pi_f$ e $\varphi_P = \varphi_Q$, allora $\varphi_Q = f$ e $Q \in \pi_f$.
  - È non banale: $\pi_f \neq \emptyset$, se f è calcolabile, e $\pi_f \neq \mathbb{U}$, perché basta scegliere una funzione f calcolabile diversa tale per cui il programma che calcola quella funzione non può appartenere a π_f.

- Sia Q un programma (MdT) e π_Q l'insieme dei programmi P, tale che si comporti come funzione calcolata uguale al calcolo di Q:

$$\pi_Q = \{P \mid \varphi_P = \varphi_Q\}$$

  Questo insieme π_Q non è ricorsivo, perché:

  - π_Q è estensionale: $P \in \pi_a$ se e solo se $\varphi_P = \varphi_Q$.
    Sia H un programma tale che $\varphi_P = \varphi_H$ e per transitività $\varphi_H = \varphi_Q$, di conseguenza $H \in \pi_Q$.
  - π_Q non è banale: $Q \in \pi_Q$, ciò implica che $\pi_Q \neq \emptyset$. Inoltre $\pi_Q \neq \mathbb{U}$, poiché esiste qualche programma diverso da Q.
  - Infine, per il teorema di Rice π_Q non è ricorsiva.

Osservazione

Attenzione al caso in cui una proprietà non sia estensionale, perché non ci sono certezze dell'insieme a cui esso appartiene. Ovvero potrebbe essere ricorsivo, ricorsivamente enumerabile oppure non ricorsivamente enumerabile. Il teorema di Rice fornisce certezze solo se la proprietà è estensionale e banale.

Lemma 5.3.1: Pudding lemma

Per ogni i e h, esiste un numero j tale che $j > h$ e $\varphi_i = \varphi_j$:

$$\forall i. \forall h. \exists j : j > h \wedge \varphi_i = \varphi_j$$

Questo lemma afferma che preso un qualunque programma, per ogni numero, esiste un programma più grande di quel numero che calcola esattamente la funzione φ_i. In pratica prendendo un programma e aggiungendo linee di codice inutili, si supera con j un qualsiasi numero h, ma algoritmicamente si calcola la stessa funzione ($\varphi_i = \varphi_j$).

Lo spazio $\{P \mid \varphi_P = \varphi_Q\}$ è la classe di equivalenza di un programma ed è infinito, numerabile e non ricorsivo.

Esercizi sulle proprietà dei programmi

Tutte le volte che si deve decidere se un programma soddisfa una proprietà estensionale, cioè se soddisfa ciò che il programma calcola, allora non è possibile avere una procedura decisionale. A volte è possibile solo dire di "sì", mentre altre volte non è possibile dire né "sì" né "no".

Un linguaggio $\mathscr{L} \subseteq \Sigma^*$ è di tipo 0, ovvero è di Turing $\mathscr{L} \in L_0$, se esiste una MdT M tale che:

$$\mathscr{L} = \{x \in \Sigma^* \mid \varphi_M(x)\downarrow\} = W_M$$

dove W_M è il dominio del programma M: valori su cui il programma è definito.

Esempio

Data una MdT M, si può definire il linguaggio $\mathscr{L} = \emptyset$?
Se il linguaggio genera l'insieme vuoto, allora è valida questa equivalenza:

$$\mathscr{L}_M = \emptyset \iff \forall x.\varphi_M(x)\uparrow$$

Si definisce la proprietà $\pi_\emptyset = \{M \mid \mathscr{L}_M = \emptyset\}$, allora si suppone che $M \in \pi_\emptyset$ e $\varphi_M = \varphi_Q$. Quest'ultima uguaglianza implica che $\mathscr{L}_Q \in \pi_\emptyset$ e $\pi_\emptyset$ sia estensionale.
Inoltre non è una proprietà banale, perché:

$$\pi_\emptyset \ni \texttt{while true \{ x = x \}}$$
$$\pi_\emptyset \not\ni \texttt{input(x); output(5);}$$

Quindi decidere che, se data una generica MdT questa definisca un linguaggio vuoto, non è possibile: per il teorema di Rice, poiché la proprietà è estensionale e non banale, allora non è decidibile (non ricorsiva).

Esempio

Data una MdT M, si può definire il linguaggio $\mathscr{L}_P = \mathscr{L}_Q$?
Fissando Q, l'insieme dei programmi che hanno lo stesso linguaggio di tipo 0 analogo al linguaggio di Q (con lo stesso dominio), è un insieme estensionale non banale. Di conseguenza chiedersi se due linguaggi di tipo 0 sono uguali, per il teorema di Rice, non è decidibile.

Teorema 5.3.10

Si suppone di considerare $A \subseteq \mathbb{N}$. Allora $|A| < \omega$ (è finito) se e solo se, per ogni $B \subseteq A$, B è ricorsivo:

$$A \subseteq \mathbb{N} : |A| < \omega \iff \forall B \subseteq A : B \in \text{REC}$$

Quindi un insieme è finito se e solo se ogni suo sottoinsieme è finito.

Dimostrazione

Si dimostrano entrambe le implicazioni:

- **Prima implicazione $\Rightarrow$**: se $|A| < \omega$, allora $\forall B \subseteq A$ con $|B| < \omega$. Ne consegue che $B \in \text{REC}$.
- **Seconda implicazione $\Leftarrow$**: si suppone che $\forall B \subseteq A$ con B ricorsivo. Per assurdo si assume che $|A| = \omega$, ma per il teorema di Cantor:

  $$\left|2^A\right| > \omega, \text{ con } \left|2^A\right| = \{x \mid x \subseteq A\}$$

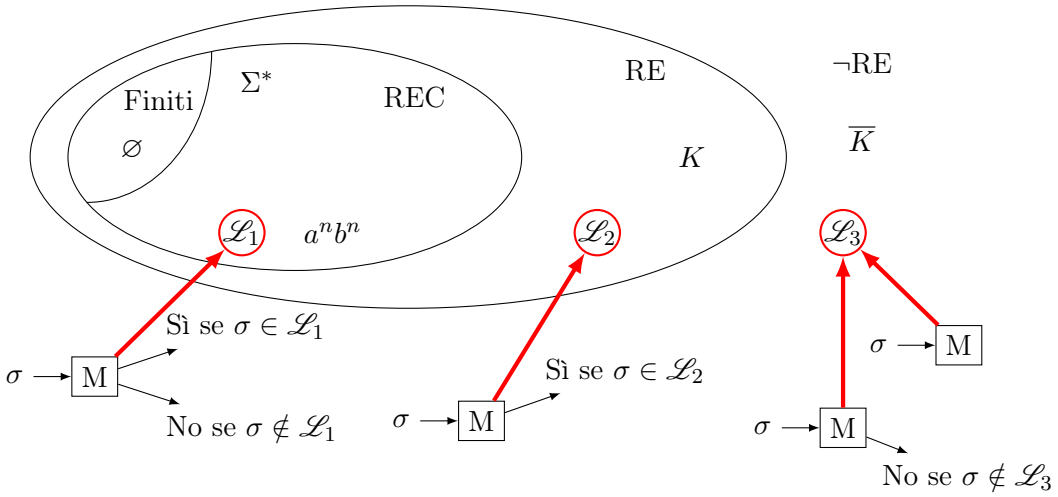
  Però questo è un assurdo, perché se A è infinito e le parti di A sono strettamente maggiori di ω, allora non è possibile richiedere che B sia un suo sottoinsieme ricorsivo: non si può enumerare l'insieme $\{x \mid x \subseteq A\}$.

Riducibilità funzionale

La difficoltà di un problema decisionale - se la stringa appartiene o meno a un linguaggio - non è determinata dalla dimensione del linguaggio.

La difficoltà è dovuta alla collocazione del linguaggio nel suo relativo insieme:

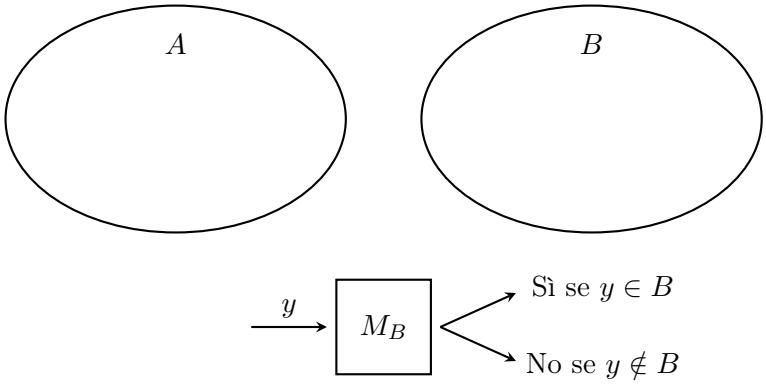
- **REC**: esiste una macchina M che permette di decidere se $\sigma \in \mathscr{L}$ oppure $\sigma \notin \mathscr{L}$.
- **RE**: esiste una macchina M che può solo determinare se $\sigma \in \mathscr{L}$.
- **¬RE**: esistono due possibilità:
  - Una macchina M in grado di decidere solo se $\sigma \notin \mathscr{L}$; questo caso è il complemento di RE.
  - Una macchina "muta" che non può affermare né $\sigma \in \mathscr{L}$ né $\sigma \notin \mathscr{L}$.



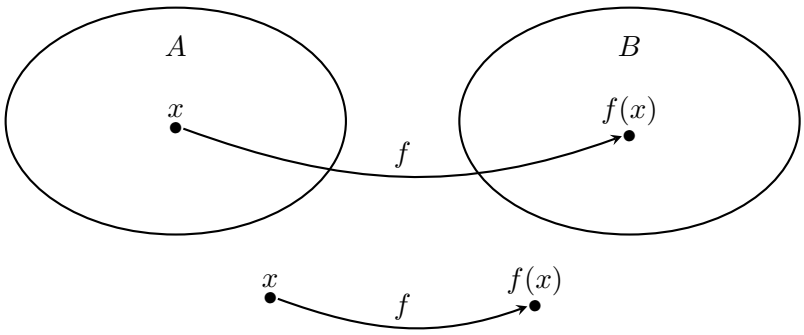
Se si considera un nuovo insieme S, dove $K \subseteq S$, allora non è possibile dedurre che $S \in$ RE. Infatti se $S = \Sigma^*$, allora $K \subseteq \Sigma^* \equiv \mathbb{N}$ dove però $\Sigma^* \in$ REC. Analogamente per $\overline{K}$ non ci sono deduzioni dirette.

6.1 Tecnica di riconoscimento

Si assume di avere due insiemi A e B, con quest'ultimo ricorsivo:



$\triangleright$ Come si trasporta[1] questa conoscenza dall'insieme B in A?
Occorre costruire una funzione tale per cui si può affermare che: un elemento sta in A se e solo se l'immagine di questa funzione sta in B.



In termini matematici questo concetto si può esprimere mediante due formule equivalenti:

$$x \in A \iff f(x) \in B \tag{6.1}$$

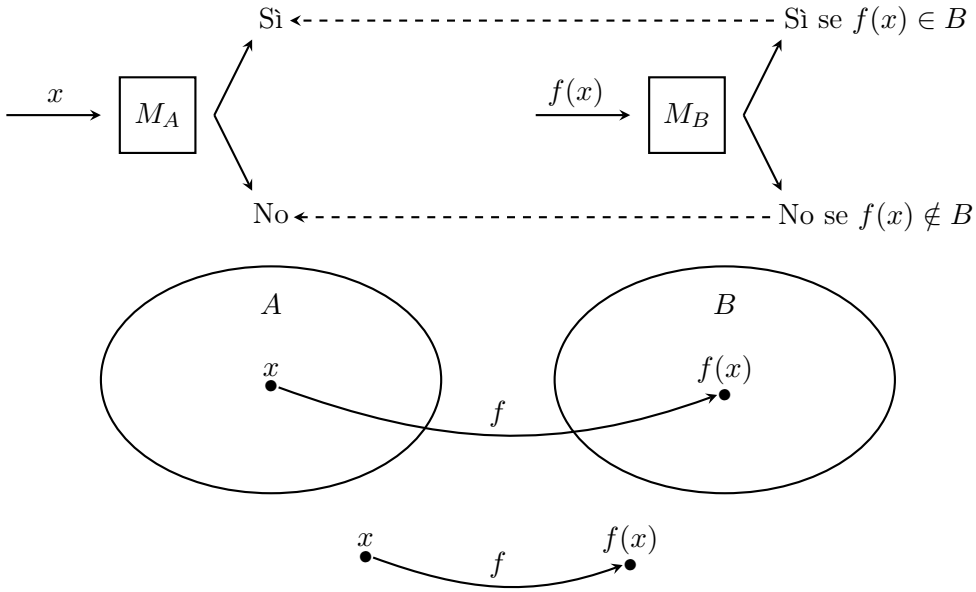
$$f(A) \subseteq B \wedge f(\overline{A}) \subseteq \overline{B} \tag{6.2}$$

dove $f(A)$ è l'immagine. Quindi sapendo che $B \in$ REC, cioè si sa decidere se "sì" o "no", per costruire la ricorsività di A bisogna:

- prendere un punto $x \in A$ in *input*;
- trasformare x, attraverso la funzione f, in un punto di B;
- assegnare il valore $f(x) \in B$ in *input* alla macchina esistente per B.

[1] Ridurre la difficoltà di B in A.

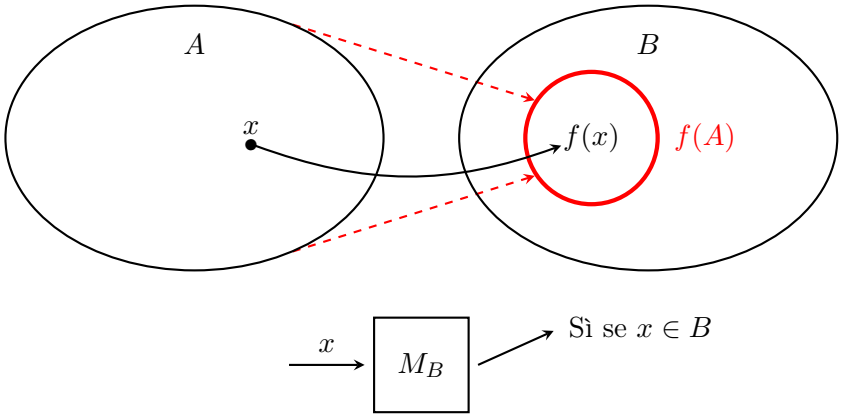
Riprendendo l'espressione (6.1) se $f(x) \in B$, allora significa che $x \in A$ e di conseguenza la macchina M_A riesce a rispondere di "sì". Analogamente se $f(x) \notin B$, allora $x \notin A$ e la macchina M_A restituisce "no" in *output*.



La funzione f deve essere totale ricorsiva, perché altrimenti se non fosse:

- **ricorsiva**: non si potrebbe ricondurre algoritmicamente la trasformazione di x in $f(x)$;
- **totale**: supponendo che sia parziale, allora per alcune x la funzione divergerebbe e di conseguenza non si potrebbe affermare il "no".

▷ Cosa succede se si suppone $B \in$ RE?



Conoscendo la relazione (6.1), si prende un punto $x \in A$ e lo si trasforma in un valore $f(x) \in B$. Se la macchina M_B restituisce "sì", allora M_A mostra "sì" in *output*, altrimenti entrambe non riescono ad affermare il "no".

6.2 Relazione di riducibilità

Definizione 6.2.1: Riducibilità funzionale $\preceq$

A si riduce funzionalmente, attraverso la funzione f, a B se f è totale ricorsiva e inoltre x appartiene ad A se e solo se $f(x)$ appartiene a B:

$$A \preceq_f B \Rightarrow f \text{ totale ricorsiva } \wedge\ x \in A \iff f(x) \in B$$

Esempio

Viene dato il seguente insieme ricorsivamente enumerabile:

$$K_2 = \{\langle x, y\rangle \mid \varphi_x(y)\downarrow\} = \{\langle x, y\rangle \mid y \in W_x\}$$

È interno all'insieme RE poiché:

$$\varphi_P(\langle x, y\rangle) = \begin{cases} 1 & \text{se } \varphi_x(y)\downarrow \\ \uparrow & \text{altrimenti} \end{cases}$$

```
1 input(x)
2 input(y)
3 if φ_INT(x,y)↓ then output(1)
```

Quindi $x \in K \Leftrightarrow x \in W_x \Leftrightarrow \langle x, x\rangle \in K_2$. Se si costruisce una funzione totale calcolabile (ricorsiva) che prende in *input* x e restituisce la coppia $\langle x, x\rangle$:

$$\underbrace{\lambda x.\langle x, x\rangle}_{f}$$

allora in questo modo si riduce funzionalmente mediante f:

$$K \preceq_f K_2$$
$$x \in K \iff f(x) = \langle x, x\rangle \in K_2$$

Se però si assume che $K_2 \in$ REC, allora:

$$f(x) = \begin{cases} 1 & \text{se } \langle x, x\rangle \in K_2 \\ 0 & \text{se } \langle x, x\rangle \notin K_2 \end{cases}$$

Questa funzione è totale ricorsiva e ciò implica che K sia ricorsivo, perché $f(x)$ è la funzione caratteristica di K, ma è un assurdo.

Esempio

Sia $A \in \text{RE}$, si mostra che $A \preceq_f K_2$; ovvero tutti gli insiemi ricorsivamente enumerabili si riducono a K_2.
Allora esiste un numero n_0 tale che A sia uguale al dominio W_{n_0}:

$$A \in \text{RE} \Rightarrow \exists n_0 : A = W_{n_0}$$

Si prende come funzione totale ricorsiva:

$$f = \lambda x.\langle n_0, x\rangle$$

Succede che x appartiene ad A se e solo se $x \in W_{n_0}$, a causa della definizione di A.
Tutto questo risulta vero se e solo se la coppia $\langle n_0, x\rangle$ appartiene a K_2:

$$x \in A \iff x \in W_{n_0} \iff \underbrace{\langle n_0, x\rangle}_{f(x)} \in K_2$$

Dunque ogni insieme ricorsivamente enumerabile è riducibile a K_2.

Da quest'ultimo esempio si dice che K_2 è completo in RE.

Alcune importanti proprietà della riduzione sono:

- **Riflessività**: A si riduce funzionalmente attraverso f in A:

$$A \preceq_f A$$

- **Transitività**: A si riduce funzionalmente attraverso f in B e B si riduce funzionalmente attraverso g in C, con f, g totali ricorsive:

$$A \preceq_f B,\ B \preceq_g C : A \preceq_{g \circ f} C$$

6.2.1 Proprietà della completezza e riduzione funzionale

La **riduzione funzionale** permette di correlare un insieme, per il quale si conosce la procedura di decisione o semi-decisione, con un insieme di cui non si conosce la difficoltà dal punto di vista della calcolabilità. Quindi si riduce un problema noto a un problema di cui si vuole cercare di comprendere la difficoltà.

Esistono alcuni insiemi, detti **insiemi completi**, per i quali tutti i ricorsivamente enumerabili sono riducibili a questi insiemi. Esiste un particolare insieme K_2 che può attrarre, secondo la riduzione funzionale, tutti gli insiemi ricorsivamente enumerabili.

Definizione 6.2.2: Completezza di un insieme

L'insieme A ricorsivamente enumerabile è completo se per ogni B ricorsivamente enumerabile questi sono riconducibili funzionalmente ad A:

$$A \in \text{RE è completo} \Rightarrow \forall B \in \text{RE} : B \preceq A$$

Corollario 6.2.1

Se A è completo e $A \equiv B$ (sono equivalenti), allora B è completo, con $B \in \text{RE}$.

Teorema 6.2.1: Completezza dell'insieme K

L'insieme $K = \{x \mid \varphi_x(x)\downarrow\} = \{x \mid x \in W_x\}$ è completo.

Dimostrazione

Sapendo che $K \in \text{RE}$, allora $K_2 \preceq K$; l'insieme dei programmi che per un certo *input* terminano (K_2) è riducibile funzionalmente all'insieme dei programmi che per *input* loro stessi terminano (K).
Per dimostrare che $K_2 \preceq K$, è necessario ricordare che K_2 lavora con coppie, mentre K lavora con valori singoli. Pertanto la funzione f deve rispettare i seguenti punti:

- Se la coppia $\langle x, y\rangle$ è tale per cui il programma x con *input* y termina, allora il programma trasformato su sé stesso termina.
- Se la coppia $\langle x, y\rangle$ non termina, allora nemmeno il programma trasformato deve terminare su sé stesso.

Le funzioni di riduzione hanno `#argomenti` $+ 1$, ossia i parametri fissati dell'*input* più il programma da dover costruire.

$$\underbrace{\text{ennupla}}_{n} \overset{\text{riduzione}}{\longrightarrow} 1 \text{ valore}$$
$$\Rightarrow \text{Programma}(\underbrace{\text{ennupla, nuovo programma}}_{n+1})$$

dove il nuovo programma è l'unica variabile.
Se invece si vuole passare a m valori, allora il programma costruito possiederà $n+m$ parametri.
Dunque si definisce il seguente programma avente come *input* la coppia $\langle x, y\rangle$ (fissati) e inoltre un ulteriore terzo *input* (variabile):

$$\varphi_P(x, y, z) = \begin{cases} 1 & \text{se } \varphi_x(y)\downarrow \\ \uparrow & \text{altrimenti} \end{cases}$$

Questa è una funzione parziale ricorsiva. Ne consegue, per SMN, che se si fissano x, y, allora si ottiene la funzione $\varphi_{S(P,x,y)}(z)$.

- **Primo caso interno a K_2**:

$$\begin{aligned}\langle x,y\rangle \in K_2 \iff \varphi_x(y)\downarrow &\Rightarrow \forall z.\varphi_{S(P,x,y)}(z) = 1\downarrow \\ &\Rightarrow \varphi_{S(P,x,y)}(S(P,x,y))\downarrow \\ &\Rightarrow S(P,x,y) \in K\end{aligned}$$

- **Secondo caso esterno a K_2**:

$$\begin{aligned}\langle x,y\rangle \notin K_2 \iff \varphi_x(y)\uparrow &\Rightarrow \forall z.\varphi_{S(P,x,y)}(z)\uparrow \\ &\Rightarrow \varphi_{S(P,x,y)}(S(P,x,y))\uparrow \\ &\Rightarrow S(P,x,y) \notin K\end{aligned}$$

Teorema 6.2.2

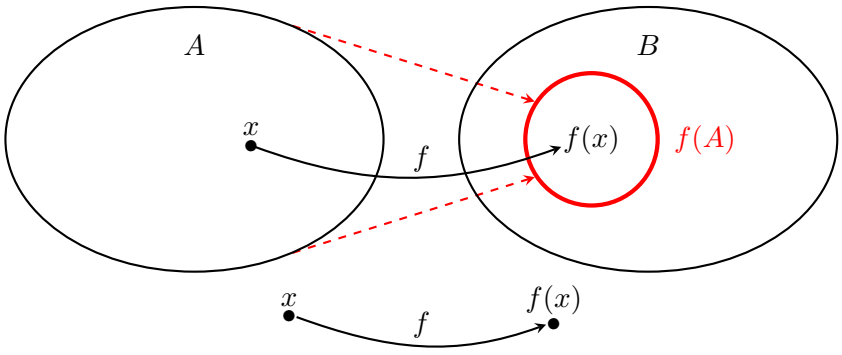
Se $A \preceq_f B$, questo è vero se solo se $\overline{A} \preceq_f \overline{B}$:

$$A \preceq_f B \iff \overline{A} \preceq_f \overline{B}$$

Dimostrazione

Si suppone che $x \in A \Leftrightarrow f(x) \in B$, con f totale ricorsiva. Ciò significa $x \notin A \Leftrightarrow f(x) \notin B$ e di conseguenza anche $x \in \overline{A} \Leftrightarrow f(x) \in \overline{B}$, che si riduce a $\overline{A} \preceq_f \overline{B}$.

Dunque se si hanno due insiemi A e B, allora in base alla funzione f l'immagine di tutto l'insieme A diventa un sottoinsieme stretto di B:



Teorema 6.2.3

Se l'insieme A si riduce funzionalmente a B, con $B \in \text{RE}$ oppure $B \in \text{REC}$, allora anche $A \in \text{RE}$ oppure $A \in \text{REC}$:

$$A \preceq B \wedge B \in \text{RE} \Rightarrow A \in \text{RE}$$
$$A \preceq B \wedge B \in \text{REC} \Rightarrow A \in \text{REC}$$

In pratica la ricorsività o la ricorsiva enumerabilità vengono ereditate per riduzione funzionale da destra verso sinistra: se si conosce la difficoltà (la struttura) del problema per B, allora per riduzione funzionale si eredita su A:

$$\overleftarrow{A \preceq B}$$

Dimostrazione

Sia l'insieme A riducibile funzionalmente verso l'insieme B, con f totale ricorsiva e $B \in \text{RE}$, dove la funzione semi-caratteristica f_B è parziale ricorsiva:

$$f_B(x) = \begin{cases} 1 & \text{se } x \in B \\ \uparrow & \text{altrimenti} \end{cases}$$

La funzione semi-caratteristica di A è definita come:

$$f_A(x) = f_B(f(x))$$

- Se f_B è parziale ricorsiva ed essendo f totale, allora implica che f_A è parziale ricorsiva: se f_B termina, allora anche f_A termina; se f_B non termina allora nemmeno f_A termina. Ne consegue inoltre che A sia ricorsivamente enumerabile.
- Se f_B è totale ricorsiva e si considera $B \in \text{REC}$, ovvero

  $$f_B(x) = \begin{cases} 1 & \text{se } x \in B \\ 0 & \text{se } x \notin B \end{cases},$$

  allora f_A è totale ricorsiva caratteristica di A. Di conseguenza l'insieme A è ricorsivo.

Osservazione

Nel caso si abbia $A \equiv B$, con $A \in \text{RE}$ oppure $A \in \text{REC}$, allora $B \in \text{RE}$ oppure $B \in \text{REC}$. È valido anche il viceversa con $B \equiv A$:

$$A \equiv B \Rightarrow A \preceq B \wedge B \preceq A$$
$$B \equiv A \Rightarrow B \preceq A \wedge A \preceq B$$

6.2.2 Applicazioni delle funzioni di riduzione

Teorema 6.2.4

Sia l'insieme $A \in \text{RE}$ e A non banale, cioè $A \neq \emptyset$ e $A \neq \mathbb{N}$. Allora A è ricorsivo se e solo se $A \preceq \overline{A}$.

Dimostrazione

Si dimostrano entrambe le implicazioni dell'equivalenza:

- **Prima implicazione $\Rightarrow$**: si suppone mediante il teorema di Post che l'insieme A sia ricorsivo se e solo se A e $\overline{A}$ sono ricorsivamente enumerabili.
  Poiché A è ricorsivo, allora esiste la funzione caratteristica totale calcolabile di A:
  $$f_A(x) = \begin{cases} 1 & \text{se } x \in A \\ 0 & \text{se } x \notin A \end{cases}$$
  Inoltre, per il teorema di caratterizzazione di Kleene, $A = \texttt{range}(g_A)$ e $\overline{A} = \texttt{range}(g_{\overline{A}})$, dove $g_{A,\overline{A}}$ sono funzioni totali calcolabili (ricorsive). Sapendo tutto ciò si costruisce il seguente programma totale $\tilde{H}$:

```
input(x)
if f_A(x) == 1 {
  output(g_Ā(x))
} else {
  output(g_A(x))
}
```

  $\varphi_{\tilde{H}} = h$ è totale calcolabile. Partendo da A, attraverso h, si arriva in $\overline{A}$ e viceversa:
  $$A \preceq_h \overline{A}$$

- **Seconda implicazione** $\Leftarrow$: si suppone che $A \preceq_f \overline{A}$, con $A \in \text{RE}$ e non banale, cioè $A \neq \emptyset$ e $A \neq \mathbb{N}$.
  Poiché $A \preceq_f B \Leftrightarrow \overline{A} \preceq_f \overline{B}$, allora esiste anche $\overline{A} \preceq_f A$. Quest'ultima riduzione funzionale significa che:

$$\overline{A} \preceq A \begin{bmatrix} x \notin A & \iff & x \in \overline{A} \\ \Updownarrow & & \Updownarrow \\ f(x) \notin \overline{A} & \iff & f(x) \in A \end{bmatrix} A \preceq \overline{A}$$

  Pertanto occorre dimostrare che $\overline{A} \in \text{RE}$ e, per il teorema di Post, ciò implica che $A \in \text{REC}$.
  Si costruisce un programma $\tilde{H}$ tale per cui, prendendo in *input* x, se la funzione semi-caratteristica di A termina, avendo il punto interno ad A e applicandogli $f(x)$, allora $f(x)$ appartiene ad A, di conseguenza x appartiene al complemento $\overline{A}$:

```
input(x)
if f_A(f(x)) == 1 then output(1)
```

  Non è possibile avere un `else` in questo programma, perché f_A è parziale ricorsiva. Tuttavia la funzione $\varphi_{\tilde{H}} = h$, tale che h è parziale ricorsiva e inoltre:

$$h(x) = \begin{cases} 1 & \text{se } f(x) \in A \Leftrightarrow x \in \overline{A} \\ \uparrow & \text{altrimenti} \end{cases}$$

  Questa funzione semi-caratteristica di $\overline{A}$ implica che $\overline{A}$ sia ricorsivamente enumerabile e per Post, sapendo che $A \in \text{RE}$ (ipotesi) e $\overline{A} \in \text{RE}$, allora la loro somma equivale a $A \in \text{REC}$:

$$\overset{\text{Post}}{\Longrightarrow} \underbrace{A \in \text{RE}}_{\text{ipotesi}} + \overline{A} \in \text{RE} \equiv A \in \text{REC}$$

Teorema 6.2.5

Siano A e B insiemi ricorsivi non banali, cioè $A, B \neq \emptyset$ e $A, B \neq \mathbb{N}$. Allora l'insieme A è equivalente a B.

Osservazione

Gli insiemi ricorsivi non banali sono tutti equivalenti tra loro.

Dimostrazione

Le funzioni caratteristiche f_B e f_A sono totali calcolabili. Inoltre $A = \texttt{range}(g_A)$ e $B = \texttt{range}(g_B)$, con g_A e g_B totali calcolabili.
Per eseguire $A \preceq_h B$, con $h = \varphi_{\tilde{H}}$ - riduzione funzionale da A verso B -, si costruisce il seguente programma $\tilde{H}$:

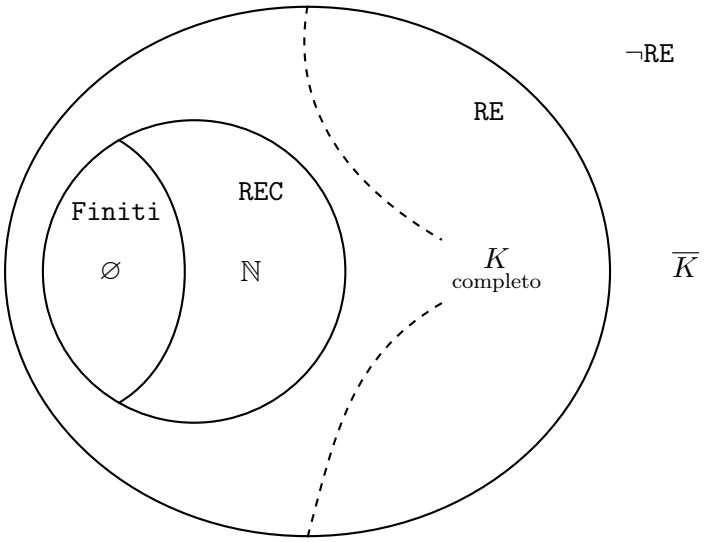
```
input(x)
if f_A(x) == 1 then output(g_B(x))
               else output(g_B̃(x))
```

Analogamente per effettuare $B \preceq_e A$, con $e = \varphi_{\tilde{E}}$ - riduzione funzionale da B verso A -, si definisce il programma $\tilde{E}$ come:

```
input(x)
if f_B(x) == 1 then output(g_A(x))
               else output(g_Ã(x))
```

Ciò dimostra che $A \equiv B$ nella riduzione funzionale.

Quindi l'attuale situazione insiemistica si presenta nel seguente modo:



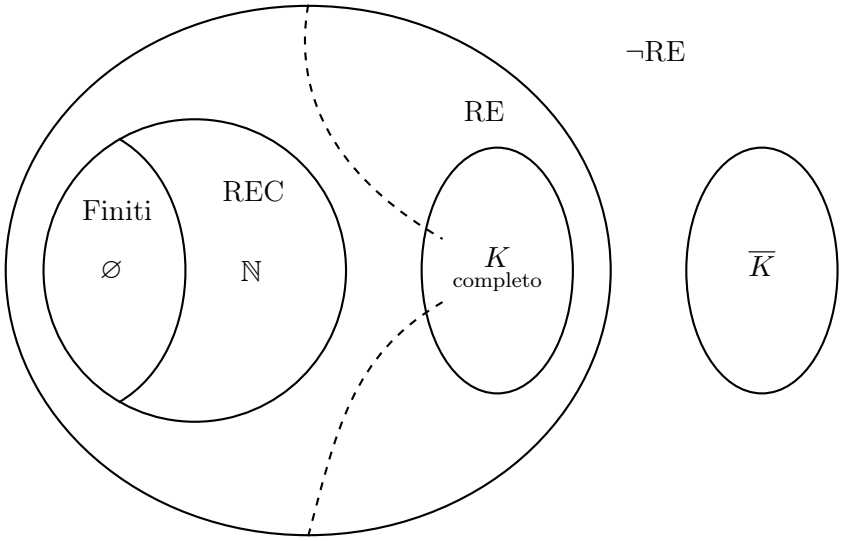
All'interno dell'insieme RE è presente K, il quale "attira" mediante la riduzione funzionale tutti i ricorsivi (REC) e i ricorsivamente enumerabili (RE), ossia K è **completo**.

Osservazione

Se $A \in \text{RE}$ e $K \preceq A$, per transitività, allora A è completo. È valida anche la relazione inversa per la quale A si riduce a K.
Perciò affermare che $A \in \text{RE}$ oppure $A \preceq K$ è equivalente.

Questa famiglia di insiemi completi non può contenere insiemi ricorsivi, poiché altrimenti si affermerebbe che i REC siano equivalenti agli RE.

Si può immaginare la famiglia degli insiemi completi interni a RE come a un'isola, che di conseguenza ha il suo complementare al di fuori di RE:



Il problema della completezza si complica nel momento in cui ci sono insiemi non ricorsivamente enumerabili, ma per cui la riduzione funzionale è possibile e di conseguenza non vanno inseriti tra gli insiemi completi.

Esempio

Viene fornito questo insieme:

$$Q = \{x \mid |W_x| < \omega\}$$

▷ Cosa succede se $K \preceq Q$?

- Se nell'osservazione precedente non fosse necessaria la condizione di ricorsiva enumerabilità, allora si potrebbe concludere che l'insieme Q sia completo.

- Intuitivamente si nota che Q non può essere ricorsivamente enumerabile. Se lo fosse, allora si riuscirebbe a rispondere con "sì" nel caso di un programma che termina solo in un insieme finito di punti. Ovviamente si possono trovare i punti di terminazione tramite il *dovetail*, ma come si può sapere che dopo un certo punto non si aggiungono ulteriori punti di terminazione?

Ignorando queste due osservazioni si procede per assurdo a eseguire $K \preceq Q$, ovvero $\overline{K} \preceq \overline{Q} = \{x \mid |W_x| = \omega\}$.
Dunque si prende un valore $x \in K$ e lo si trasforma, con $f(x)$, in un valore appartenente a Q, tale che $|W_{f(x)}| < \omega$. Commettendo un errore si potrebbe definire la seguente funzione parziale:

$$\psi(x,y) = \begin{cases} \uparrow & \text{se } x \in K \\ 1 & \text{se } x \notin K \end{cases}$$

Tuttavia questa funzione non è ricorsiva, poiché non si riesce a scrivere l'`if/else`:

```
input(x)
input(y)
if φ_INT(x,x)↓ {
  while true { x = x }
} else {
  output(1)
}
```

L'`else` non può esistere, in quanto bisogna attendere che il programma x, con *input* x, termini. Non si può affermare che non convergerà per una x qualsiasi.

$\Rightarrow$ Come si può generare, da un programma che termina su sé stesso, un programma che termina su un insieme finito di punti? Ma se il programma non termina su sé stesso, allora si genera un programma che termina sempre?
Si costruisce una funzione parziale ricorsiva in questa maniera:

$$\psi(x,y) = \begin{cases} 1 & \text{se } \varphi_x(x)\not\downarrow \text{ in meno di } y \text{ passi} \\ \uparrow & \text{altrimenti} \end{cases}$$

```
input(x)
input(y)
if φ_INT(x,x)↓ < y then output(1)
    else { while true { x = x } }
```

Dato che si ha un certo y, allora è possibile decidere se il programma termina o meno in y passi. Per SMN, fissando x, si ottiene che:

$$\psi(x, y) = \varphi_{f(x)}(y)$$

Ne consegue che questa funzione trasforma un programma per cui $\varphi_x(x)$ termina in un programma che ha un dominio finito, ma se $\varphi_x(x)$ non dovesse terminare, allora il dominio del programma $f(x)$ risulta infinito.

- Se $x \in K$, allora per definizione $\varphi_x(x)\downarrow$ se e solo se esiste un tempo $t \in \mathbb{N}$ tale che $\varphi_x(x)\downarrow$ in t passi, allora per ogni $m < t$ $\varphi_x(x)\not\downarrow$ in m passi. Ne consegue che per ogni $y \in [0, t-1]$ si ha $\varphi_{f(x)}(y) = 1$ e per ogni $y \geq t$ si ha $\varphi_{f(x)}(y)\uparrow$. Risulta infine che il dominio $W_{f(x)}$ è contenuto o uguale all'intervallo $[0, t-1]$, ovvero $f(x) \in \{x \mid |W_x| < \omega\}$.

$$\begin{aligned} x \in K \overset{\Delta}{\iff} \varphi_x(x)\downarrow &\iff \exists t \in \mathbb{N}.\varphi_x(x)\downarrow^{t \text{ passi}} \\ &\Rightarrow \forall m < t.\varphi_x(x)\not\downarrow^{m \text{ passi}} \\ &\Rightarrow \forall y \in [0, t-1].\varphi_{f(x)}(y) = 1 \wedge \forall y \geq t.\varphi_{f(x)}(y)\uparrow \\ &\Rightarrow W_{f(x)} \subseteq [0, t-1] \\ &\Rightarrow |W_{f(x)}| < \omega \equiv f(x) \in \{x \mid |W_x| < \omega\} \end{aligned}$$

- Se $x \notin K$, allora per definizione $\varphi_x(x)\uparrow$. Di conseguenza la condizione di $\psi(x, y)$, per cui restituisce 1 nel caso $\varphi_x(x)\not\downarrow$ in meno di y passi, è sempre vera per ogni $y \in \mathbb{N}$. Ciò comporta che il dominio $W_{f(x)} = \mathbb{N}$ ed è infatti infinito, causando $f(x) \notin \{x \mid |W_x| < \omega\}$.

$$\begin{aligned} x \notin K \overset{\Delta}{\iff} \varphi_x(x)\uparrow &\Rightarrow \forall y \in \mathbb{N}.\varphi_{f(x)}(y) = 1 \\ &\Rightarrow W_{f(x)} = \mathbb{N}, \quad |\mathbb{N}| = \omega \\ &\Rightarrow f(x) \notin \{x \mid |W_x| < \omega\} \end{aligned}$$

6.3 Insiemi creativi e produttivi

Definizione 6.3.3: Insieme creativo

L'insieme $A \in \text{RE}$ è creativo se il suo complemento $\overline{A}$ è produttivo:

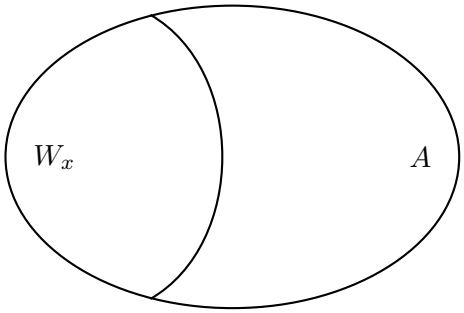
$$A \in \text{RE (Creativo)} \Rightarrow \overline{A} \in \neg\text{RE (Produttivo)}$$

Definizione 6.3.4: Insieme produttivo

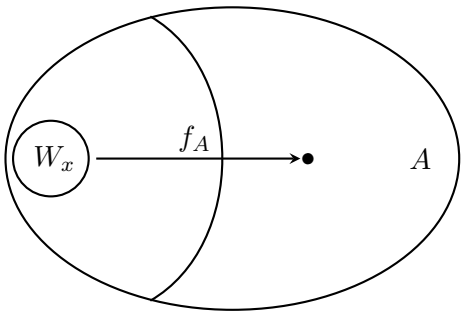
L'insieme A è produttivo se esiste una funzione totale ricorsiva f_A, tale che per ogni $x \in \mathbb{N}$ e il dominio W_x è sottoinsieme o uguale ad A, allora f_A trasforma il punto x facendolo appartenere ad A e non a W_x:

$$\forall x \in \mathbb{N} : W_x \subseteq A \Rightarrow f_A(x) \in A \setminus W_x$$

Graficamente questo significa che se si dispone di un insieme A e si prende un qualsiasi programma x, il quale cerca di enumerare tutti i punti di A, allora il suo dominio W_x è all'interno di A:



La funzione f_A totale ricorsiva è in grado di trasformare x in un punto che appartiene ad $A \setminus W_x$:



Per come si sceglie il tentativo di enumerare A, esiste sempre un punto dell'insieme A che sfugge a quel tentativo di enumerazione.

Quindi se l'insieme A risulta essere **produttivo**, allora non può essere ricoperto completamente da un qualunque programma, cioè non si riesce a enumerarlo. Esiste sempre un programma dell'insieme A che trasforma quel tentativo in un punto che sfugge.

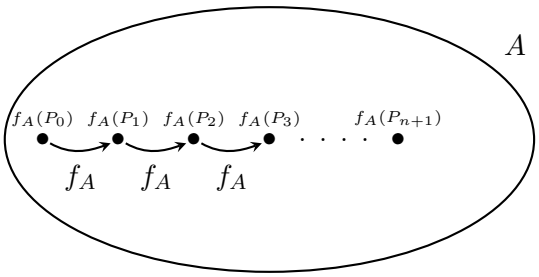
Osservazione

Un insieme è produttivo se non può mai essere enumerato.

Si considera l'insieme $A \in$ Produttivi. Allora esiste un certo programma n_0 tale per cui il suo dominio è sottoinsieme o uguale ad A, con cardinalità del dominio infinita:

$$A \in \text{Produttivi} \Rightarrow \exists n_0 . W_{n_0} \subseteq A, \quad \text{con } |W_{n_0}| = \omega$$

L'idea consiste nel generare un punto in A e di considerare il programma che termina solamente su quel punto. Questo diventa un tentativo di enumerare A; l'indice di questo programma è esterno al punto generato.
Dopodiché si prende un nuovo programma che contiene entrambi i punti. Anche questo diventa un nuovo tentativo di enumerare A e il suo indice è esterno a questi due punti. Continuando in maniera analoga, si formula un ulteriore nuovo programma che termina su questi tre punti, ma che risulta essere ancora esterno, e di conseguenza si prosegue così all'infinito.



Ogni volta, per la funzione di produttività f_A, si genera un nuovo punto e si può continuare all'infinito. Ecco che si crea una numerazione infinita che non potrà mai coprire tutto l'insieme A (perché è produttivo).

- f_A: funzione di produttività di A:

$$\forall x . W_x \subseteq A \Rightarrow f_A(x) \in A \setminus W_x$$

- Si costruisce la funzione g totale ricorsiva e iniettiva in A:

$$\texttt{range}(g) \subseteq A \wedge |\texttt{range}(g)| = \omega$$

▷ Come si costruisce il primo programma?
Si effettua una ragionevole osservazione: l'insieme vuoto è sempre contenuto in ogni insieme, perciò $\emptyset \subseteq A$ e $\emptyset$ è il dominio di un programma che non conclude mai la propria esecuzione:

$$W_{P_0} = \emptyset : \quad \texttt{while true \{ x = x \}}$$

Quindi $f_A(P_0) \in A \setminus W_{P_0} = A \setminus \emptyset = A$.
Per generare un secondo programma, diverso dal primo, che termina esattamente in quel punto, occorre usare il primo punto come dominio di un tentativo per enumerare A:

$$\{f_A(P_0)\} \subseteq A \wedge \{f_A(P_0)\} = W_{P_1}$$

Si deve costruire un programma che termina solo su $f_A(P_0)$ e diverge altrimenti. Tuttavia bisogna stare attenti:

- P_0 è un numero (l'indice del primo programma).
- f_A è una funzione totale ricorsiva. Partendo da P_0 restituisce un numero:

```
input(x)
if x == f_A(P_0) then output(1)
                 else while true { x = x }
```

Questo programma termina quando x corrisponde all'unico punto noto, cioè $f_A(P_0)$, restituendo in *output* 1. Altrimenti rimane in *loop* all'infinito (diverge). Applicando f_A al programma P_1 si genera un punto che appartiene ad A, ma non al tentativo: diverso da $f_A(P_0)$.

$$f_A(P_1) \in A \setminus W_{P_1} = A \setminus \{f_A(P_0)\} \Rightarrow f_A(P_1) \neq f_A(P_0)$$

Si prosegue in questa maniera all'infinito. Generalizzando l'intero processo si ottiene:

$$\underbrace{f_A(P_0), \ldots, f_A(P_n)}_{W_{P_{n+1}}} \subseteq A$$

```
input(x)
if x == f_A(P_0) || ... || x == f_A(P_n) {
  output(1)
} else {
  while true { x = x }
}
```

Questo programma P_{n+1} è un tentativo che enumera A, pertanto si ottiene:

$$f_A(P_{n+1}) \in A \setminus \{f_A(P_0), \ldots, f_A(P_n)\}$$

Quindi $\forall i = 0 \ldots n$: $f_A(P_i) \neq f_A(P_{n+1})$. Allora dato un insieme finito è sempre possibile costruire il programma (generare il punto) successivo.
La funzione costruita g ha come *range* questa sequenza di punti, tutti contenuti all'interno di A e diversi tra loro.

$$g = \begin{cases} g(0) = f_A(P_0) \in A \\ g(n+1) = f_A(P_{n+1}) \in A \end{cases}$$

Questa funzione è totale ricorsiva, dove $\texttt{range}(g) \subseteq A$ e $|\texttt{range}(g)| = \omega$. Ponendo $W_{n_0} = \texttt{range}(g)$ si sa, per il teorema di caratterizzazione degli insiemi ricorsivamente enumerabili di Kleene, che un insieme è RE se e solo se è l'*output* di una funzione totale ricorsiva. Quindi ogni insieme produttivo contiene al suo interno un insieme RE infinito.

Teorema 6.3.6

L'insieme K è creativo, di conseguenza $\overline{K}$ è produttivo.

Dimostrazione

L'insieme $K \in \text{RE}$, poiché il suo programma $\tilde{P}$ consiste in:

```
input(x)
if φ_INT(x,x)↓ then output(1)
```

con $K = W_{\tilde{P}}$. L'insieme $\overline{K}$ è produttivo, perché esiste una funzione $f_{\overline{K}}$ totale ricorsiva tale che se $W_x \subseteq \overline{K}$, allora $f_{\overline{K}}(x)$ appartiene a $\overline{K}$ e non a W_x:

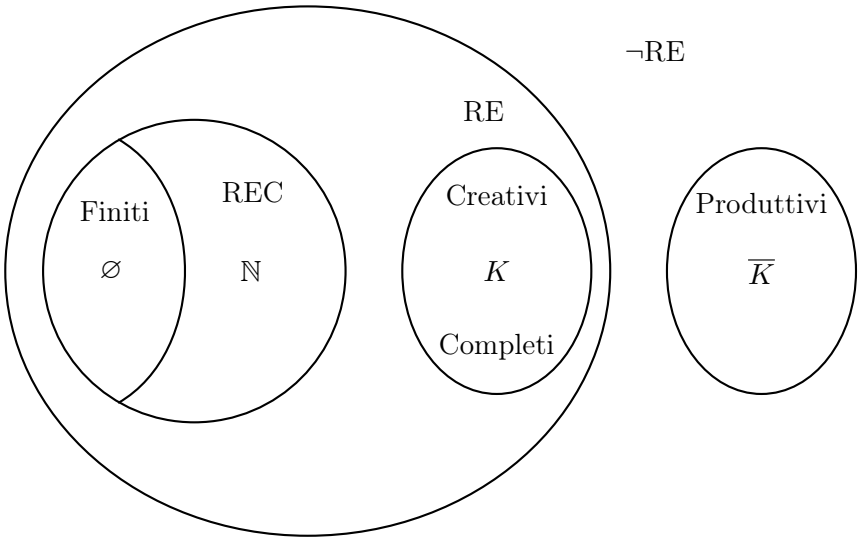
$$W_x \subseteq \overline{K} \Rightarrow f_{\overline{K}}(x) \in \overline{K} \setminus W_x$$

Nel dettaglio si sceglie come funzione di produttività dell'insieme $\overline{K}$ la funzione identità $f_{\overline{K}} = \lambda x.x$; essendo primitiva ricorsiva è di conseguenza totale ricorsiva.
Si suppone che $W_x \subseteq \overline{K}$. Se si sostituisce la funzione identità a $f_{\overline{K}}$, allora si dimostra che x appartiene a $\overline{K}$ e non a W_x:

$$\begin{aligned} W_x \subseteq \overline{K} &\Rightarrow f_{\overline{K}}(x) \in \overline{K} \setminus W_x \\ &\Rightarrow x \in \overline{K} \setminus W_x \end{aligned}$$

Ipotizzando che $x \in W_x$, allora per definizione $x \in K = \{x \mid x \in W_x\}$. Tuttavia per ipotesi è noto che $W_x \subseteq \overline{K}$ e ciò implica che $x \in \overline{K}$. Pertanto se fosse vera questa ipotesi, x apparterrebbe sia a un insieme sia al suo complemento: assurdo.
In conclusione $x \notin W_x$ e ciò comporta che $x \in \overline{K} = \{x \mid x \notin W_x\}$.

In pratica questo teorema afferma che se si prova a enumerare, ovvero generare, con un algoritmo tutti i programmi che non terminano su loro stessi, allora è proprio quel programma che enumera a non essere enumerato da sé stesso.



Dato un insieme A:

- Se è finito, allora $A \in \text{REC}$.
- Se $A, \overline{A} \in \text{RE}$, allora $A \in \text{REC}$.
- Se $A \in \text{Completi/Creativi(RE)}$, allora $\overline{A} \in \text{Produttivi}(\neg\text{RE})$.
- Se $A \in \text{Produttivi}(\neg\text{RE})$, allora $\overline{A} \in \text{Produttivi}(\neg\text{RE})$ oppure $\overline{A} \in \text{Creativi(RE)}$.

Osservazione

Se un insieme è creativo, allora il suo complemento è produttivo. ma non vale sempre il viceversa: il complemento di un produttivo può essere o no un creativo. Quindi l'insieme dei produttivi è esattamente speculare all'insieme dei creativi, non il viceversa.

Teorema 6.3.7

Siano $A, B \subseteq \mathbb{N}$, allora:

1. Se l'insieme A è produttivo, allora $A \in \neg\text{RE}$.
2. Inoltre se $A \preceq B$, allora l'insieme B è produttivo.

$$A \in \text{Produttivi} \Rightarrow A \in \neg\text{RE}$$
$$A \in \text{Produttivi} \wedge A \preceq B \Rightarrow B \in \text{Produttivi}$$

Dunque i due casi esistenti sono:

- La ricorsiva enumerabilità si propaga verso sinistra:
$$\overleftarrow{A \preceq B} \wedge B \in \text{RE} \Rightarrow A \in \text{RE}$$
- La non ricorsiva enumerabilità si propaga da sinistra verso destra:
$$\overrightarrow{A \preceq B} \wedge A \in \text{Produttivi} \Rightarrow B \in \text{Produttivi}$$

Dimostrazione

Si dimostrano entrambi i punti del teorema:

- **Punto (1)**: per ogni x (qualunque programma), se W_x è un sottoinsieme o uguale ad A, allora $f_A(x)$ appartiene ad A e non a W_x:
$$\forall x. W_x \subseteq A \Rightarrow f_A(x) \in A \setminus W_x$$
Se $A \in \text{RE}$, allora esiste un n_0 tale per cui $A = W_{n_0}$. Applicando la produttività si ottiene $f_A(n_0) \in A \setminus W_{n_0}$, ma poiché $A = W_{n_0}$, allora $A \setminus W_{n_0} = \emptyset$

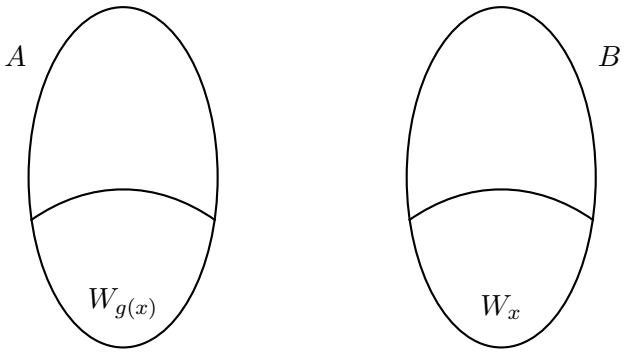
e cioè assurdo: un punto non può esistere in un insieme vuoto. Ne consegue che A non può essere ricorsivamente enumerabile.

- **Punto (2)**: per ipotesi si suppone che $A \in$ Produttivi e $A \preceq_f B$. È noto che ogni punto di A lo si può mappare in un punto interno a B, mentre ogni punto al di fuori di A lo si può mappare in un punto esterno a B, tramite la funzione f. Bisogna dimostrare che B sia un insieme produttivo.
  Per farlo si prende $W_x \subseteq B$, cioè si cerca di enumerare con W_x l'insieme B. Allora viene definita la seguente funzione parziale ricorsiva:

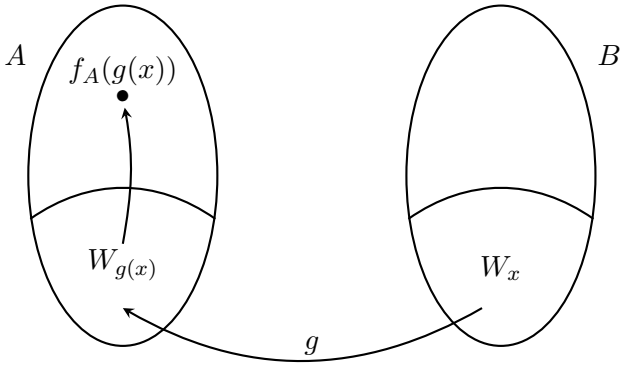
$$\psi(x,y) = \begin{cases} 1 & \text{se } f(y) \in W_x \\ \uparrow & \text{altrimenti} \end{cases}$$

```
input(x)
input(y)
y = f(y)
if φ_INT(x,y)↓ then output(1)
```

  Per SMN $\psi(x,y) = \varphi_{g(x)}(y)$, con g totale ricorsiva e $W_{g(x)} = \{y \mid \varphi_{g(x)}(y)\downarrow\}$ $= \{y \mid f(y) \in W_x\}$: sono tutte le y tali che con la funzione di riduzione f si vada dentro W_x. Inoltre l'insieme degli y per cui $f(y) \in W_x$, nell'ipotesi in cui $W_x \subseteq B$, significa che $W_{g(x)} \subseteq A$.
  Questo è dovuto al fatto che con la funzione di riduzione si va all'interno di B se e solo se si parte dall'insieme A: $f(y) \in B \Leftrightarrow y \in A$.

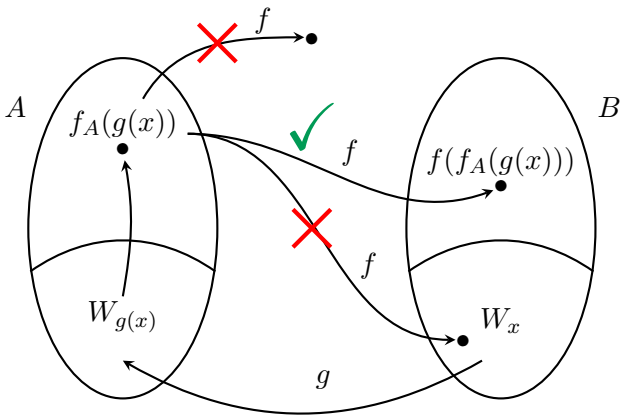


  Si ricorda che per ipotesi l'insieme A è produttivo.
  Quanto fatto fin'ora diventa un tentativo di enumerare l'insieme A e la funzione di produttività f_A deve portare ogni tentativo di enumerare l'insieme A in un punto esterno a quel tentativo, ma pur sempre appartenente ad A.
  Pertanto se $g(x)$ prende x (tentativo di enumerare B) e lo trasforma in un tentativo di enumerare A, allora diventa di conseguenza un tentativo di enumerare un insieme produttivo. Perciò, attraverso la funzione di produttività f_A, si porta il punto $g(x)$ a essere un punto esterno a $W_{g(x)}$, ma ancora interno ad A.



Questo punto appena trovato non può stare all'interno di $W_{g(x)}$, perché altrimenti l'insieme A non sarebbe produttivo. Adesso occorre tornare all'interno di B e applicando la funzione f al punto $f_A(g(x))$ si hanno tre possibilità:

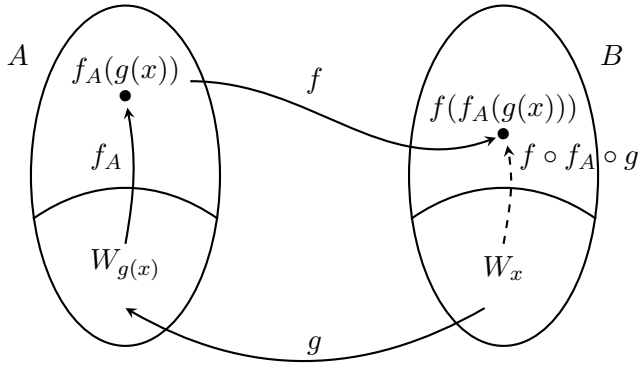
1. Andare fuori sia da A sia da B, ma è impossibile. Perché la funzione di riduzione quando parte da un punto di A finisce dentro B.
2. Andare in W_x, ma non è possibile. Perché $W_{g(x)}$ è uguale all'insieme di y tali per cui $f(y) \in W_x$, di conseguenza si avrebbe che $f_A(g(x)) \in W_{g(x)}$, ma per produttività questo punto $f_A(g(x)) \notin W_{g(x)}$.
3. Andare in B, ma fuori da W_x, è l'unica possibilità valida.



$f(f_A(g(x)))$ è la composizione di tre funzioni totali ricorsive:

- g è totale ricorsiva per SMN;
- f_A è totale ricorsiva per produttività di A;
- f è totale ricorsiva per riducibilità funzionale di A a f;

poiché la composizione di tre funzioni totali ricorsive è totale ricorsiva, allora la funzione di produttività di B risulta essere $f \circ f_A \circ g$:



Dato che W_x è generico, ne consegue che B sia un insieme produttivo.

Corollario 6.3.2

Se l'insieme A è creativo, allora ne consegue che non sia ricorsivo:

$$A \in \text{Creativi} \Rightarrow A \notin \text{REC}$$

Dimostrazione

Se l'insieme A è creativo, allora il suo complemento $\overline{A}$ è produttivo. Ne consegue a sua volta che $\overline{A} \in \neg\text{RE}$ e per il teorema di Post $A \notin \text{REC}$.

Corollario 6.3.3

Se l'insieme A è creativo e $A \preceq B$, allora se $B \in \text{RE}$ ne consegue che B sia un insieme creativo:

$$A \in \text{Creativi} \wedge B \in \text{RE} \wedge A \preceq B \Rightarrow B \in \text{Creativi}$$

Dimostrazione

Si suppone che gli insiemi siano $A \in \text{Creativi}$ e $B \in \text{RE}$, dopodiché si ipotizza anche $A \preceq B$ che equivale ad affermare $\overline{A} \preceq \overline{B}$. Inoltre poiché $A \in \text{Creativi}$, equivale al fatto che $A \in \text{RE}$ e $\overline{A} \in \text{Produttivi}$.

$$A \preceq B \iff \overline{A} \preceq \overline{B}$$
$$A \in \text{Creativi} \iff A \in \text{RE} \wedge \overline{A} \in \text{Produttivi}$$

Entrambi i risultati delle equivalenze, ossia $\overline{A} \preceq \overline{B}$ e $\overline{A} \in \text{Produttivi}$, implicano che

l'insieme $\overline{B}$ sia produttivo:

$$\overline{A} \preceq \overline{B} \wedge \overline{A} \in \text{Produttivi} \Rightarrow \overline{B} \in \text{Produttivi}$$

Sapendo che $B \in \text{RE}$ (per ipotesi) e $\overline{B} \in$ Produttivi (risultato appena ricavato), ne consegue che B sia un insieme creativo:

$$B \in \text{RE} \wedge \overline{B} \in \text{Produttivi} \iff B \in \text{Creativi}$$

6.3.1 Equivalenza dell'insieme creativo e completo

Un insieme A, se è completo, allora è anche creativo e viceversa. Dunque non c'è alcuna differenza nel definire un insieme A creativo oppure completo.

Teorema 6.3.8: Equivalenza tra insiemi completi e creativi (John Myhill)

Un insieme A è ricorsivamente enumerabile completo se e solo se A è creativo:

$$A \in \text{Completi} \iff A \in \text{Creativi}$$

Dimostrazione

Si dimostrano entrambe le implicazioni sfruttando il secondo teorema di ricorsione di Kleene:

- **Prima implicazione** $\Rightarrow$: sia A un insieme ricorsivamente enumerabile completo, ciò equivale ad affermare che ogni $x \in \text{RE}$ si riduce funzionalmente ad A:

  $$A \in \text{Completi(RE)} \iff \forall x \in \text{RE} : x \preceq A$$

  Poiché l'insieme $K \in \text{RE}$, allora K si riduce all'insieme A:

  $$K \in \text{RE} \Rightarrow K \preceq A$$

  Quest'ultima riduzione risulta vera se e solo se $\overline{K}$ si riduce funzionalmente ad $\overline{A}$:

  $$K \preceq A \iff \overline{K} \preceq \overline{A}$$

  Per il secondo corollario $\overline{K} \preceq \overline{A}$ implica che $\overline{A}$ sia produttivo. Di conseguenza sapendo che $A \in \text{RE}$ e $\overline{A} \in$ Produttivi, per definizione equivale al fatto che A sia un insieme creativo:

  $$A \in \text{RE} \wedge \overline{A} \in \text{Produttivi} \iff A \in \text{Creativi}$$

- **Seconda implicazione** $\Leftarrow$: sia per ipotesi A un insieme creativo, allora per definizione $A \in \text{RE}$ e $\overline{A} \in$ Produttivi. Successivamente si prende $f_{\overline{A}}$ (funzione totale ricorsiva produttiva di $\overline{A}$) e un insieme generico $B \in \text{RE}$. Quindi occorre dimostrare che $B \preceq A$, per poter derivare $B \in \text{Completi}(RE)$, costruendo la

funzione di riduzione da un generico B ad A.
Allora si definisce la seguente funzione parziale ricorsiva:

$$\psi(x,y,z) = \begin{cases} 0 & \text{se } y \in B \text{ e } z = f_{\overline{A}}(x) \\ \uparrow & \text{altrimenti} \end{cases}$$

Questa funzione mostra in *output* 0 se con l'*input* y il programma termina, cioè il punto y appartiene all'insieme B, e se z corrisponde al valore calcolato da $f_{\overline{A}}$ su x. Il programma è il seguente:

```
1 input(x,y,z)
2 if z == f_Ā(x) then {
3   if y∈B then output(0)
4 } else while true { x = x }
```

> Non si può inserire un `else` alla condizione $y \in B$, perché bisogna attendere che la macchina, che enumera B, termini. Se questa macchina termina su y, allora $y \in B$.

Questo programma calcola la funzione ψ e fissando x, y per SMN:

$$\psi(x,y,z) = \varphi_{g(x,y)}(z)$$

▷ Che dominio ha $W_{g(x,y)}$?

$$W_{g(x,y)} = \begin{cases} \{f_{\overline{A}}(x)\} & \text{se } y \in B \\ \{\emptyset\} & \text{se } y \notin B \end{cases}$$

> Questa non è una funzione, ma è la descrizione matematica del dominio del programma avendo fissato x, y.

$W_{g(x,y)}$ è la descrizione matematica dell'insieme delle z tali per cui il programma termina. Questo è possibile in quanto:

- se $y \in B$, allora il dominio del programma è $\{f_{\overline{A}}(x)\}$;
- se $y \notin B$, allora il dominio del programma è l'insieme vuoto.

Essendo g una funzione totale ricorsiva per SMN, allora tramite il secondo teorema di ricorsione di Kleene ne consegue che: esiste una funzione totale ricorsiva v, tale per cui per ogni y $\varphi_{g(v(y),y)}$ è uguale a $\varphi_{v(y)}$.

$$\Rightarrow \exists v(\text{totale ricorsiva}) : \forall y.\varphi_{g(v(y),y)} = \varphi_{v(y)}$$

Ovvero per ogni y $W_{(g(v),y)}$ è uguale a $W_{v(y)}$ già definita in precedenza:

$$\forall y.W_{(g(v),y)} = W_{v(y)} = \begin{cases} \{f_{\overline{A}}(v(y))\} & \text{se } y \in B \\ \{\emptyset\} & \text{se } y \notin B \end{cases}$$

Pertanto il teorema del punto fisso mostra l'esistenza di una trasformazione v (totale ricorsiva) che è punto fisso della funzione g. In questo modo si riesce a ottenere un programma $v(y)$ che possiede lo stesso dominio del programma $g(v(y), y)$.
Ecco che si dimostra la riducibilità funzionale da B ad A con la funzione $f_{\overline{A}} \circ v$:

- **Da B si va dentro ad A**: se $y \in B$, allora $W_{v(y)} = \{f_{\overline{A}}(v(y))\} \subseteq A$. Se si ipotizza che $f_{\overline{A}}(v(y)) \notin A$, ossia $\{f_{\overline{A}}(v(y))\} = W_{v(y)} \subseteq \overline{A}$, allora diventa un tentativo di enumerare $\overline{A}$ e ciò implica che: utilizzando la funzione di produttività $f_{\overline{A}}$ al tentativo di enumerazione, questo appartenga ad $\overline{A}$ meno $W_{v(y)}$:

  $$\{f_{\overline{A}}(v(y))\} = W_{v(y)} \subseteq \overline{A} \Rightarrow f_{\overline{A}}(v(y)) \in \overline{A} \setminus W_{v(y)}$$

  Tuttavia ciò è assurdo: il punto $\{f_{\overline{A}}(v(y))\}$ è uguale a $W_{v(y)}$; quindi $f_{\overline{A}}(v(y)) \in A$.
- **Da fuori di B si va fuori ad A**: se $y \notin B$, allora $W_{v(y)} = \emptyset$, il quale è contenuto sempre da ogni insieme e perciò $\emptyset \subseteq \overline{A}$. Questo diventa un tentativo di enumerare l'insieme $\overline{A}$. Infatti, se $\overline{A} \in$ Produttivi, il programma che ha come dominio l'insieme vuoto (per esempio `while true { x = x }`) viene trasformato in un punto di $\overline{A}$ che non è catturato da questo programma. Di conseguenza la funzione di produttività $f_{\overline{A}}$, sul tentativo $v(y)$, appartiene ad $\overline{A}$ meno $W_{v(y)}$:

  $$\begin{aligned} &\Rightarrow f_{\overline{A}}(v(y)) \in \overline{A} \setminus W_{v(y)} \\ &\Rightarrow f_{\overline{A}}(v(y)) \in \overline{A} \setminus \emptyset = \overline{A} \end{aligned}$$

  Questo implica che, essendo $f_{\overline{A}} \circ v$ totale ricorsiva, l'insieme B si riduce funzionalmente ad A con $f_{\overline{A}} \circ v$.

Perciò A è un insieme completo, perché si è scelto un generico insieme B su cui effettuare questa operazione.