

Fondamenti di informatica

UniVR - Dipartimento di Informatica

Fabio Irimie

1° Semestre 2025/2026

Indice

1 Introduzione	3
1.1 Cos'è l'informatica?	3
1.2 Origini dell'informatica	3
1.2.1 Calcolabilità	3
1.3 Nozioni di base	3
1.3.1 Basi di logica	3
1.3.2 Nozioni sugli insiemi	4
1.3.3 Nozioni sulle relazioni	5
1.3.4 Nozioni sulle funzioni	5
1.4 Funzioni calcolabili	6
1.4.1 Quante funzioni numerabili ci sono?	6
1.4.2 Funzioni vs Insiemi	8
2 Linguaggi regolari	9
2.1 Linguaggi formali	11
2.2 Linguaggi regolari (automi a stati finiti, DFA)	12
2.2.1 Come si dimostra che un linguaggio è regolare?	13
2.3 Automi a stati finiti non deterministici (NFA)	18
2.3.1 Linguaggio riconosciuto da un NFA	19
2.3.2 Conversione da NFA a DFA	21
2.4 Automi non deterministici con ϵ -transizioni (ϵ -NFA)	22
2.4.1 ϵ -closure	23
2.5 Espressioni regolari	24
2.6 Proprietà dei linguaggi regolari	26
2.6.1 Proprietà di chiusura	26
2.6.2 Proprietà di decidibilità	27
2.6.3 Esistenza dell'automa minimo	27
2.6.4 Condizione necessaria perché un linguaggio sia regolare	32
2.6.5 Dimostrazione che un linguaggio non è regolare	34
3 Linguaggi context free	37
3.1 Grammatiche context free	37
3.1.1 Dimostrazione canonica che un linguaggio è context free	39
3.1.2 Dimostrazione alternativa	43
3.2 Alberi di derivazione	43
3.3 Ambiguità delle grammatiche	44
3.4 Forma normale	45
3.5 Conversione in forma normale di Chomsky	45
3.5.1 Eliminazione dei simboli inutili	46
3.5.2 Eliminazione delle produzioni unitarie	47
3.5.3 Eliminazione delle ϵ -produzioni	48
3.6 Trasformazione di una grammatica in forma normale di Chomsky	49
3.7 Transformazione in forma normale di Greibach	49
3.8 Pumping lemma per i linguaggi context free	50
3.8.1 Dimostrazione (grafica)	51
3.8.2 Dimostrare che un linguaggio non è context free	51
3.9 Proprietà di chiusura dei linguaggi context free	53
3.10 Problemi decidibili per i linguaggi context free	55

3.11	Automa per riconoscere i linguaggi context free (Automi a pila)	55
3.12	Corrispondenza tra APND e grammatiche context free	58
4	Teoria della calcolabilità	59
4.1	Potenza espressiva	59
4.2	Teoria della Calcolabilità	60
4.3	Funzioni primitive ricorsive	61
4.3.1	Funzioni primitive di base	61
4.3.2	Operazioni sulle funzioni	61
4.3.3	Esempi	62
4.3.4	Calcolabilità delle funzioni primitive ricorsive	64
4.4	Macchina di Turing	65
4.4.1	Descrizione istantanea di una MdT	66
4.5	Funzioni parziali ricorsive	69
4.5.1	Aritmetizzazione delle macchine di Turing	73
4.5.2	Fattorizzazione di Gödel	74
4.5.3	Macchina universale (interprete)	75
4.6	Lambda calcolo	76
4.7	Problemi insolubili	76
4.7.1	Problema della terminazione	76
4.8	Specializzatore	78
4.9	Prima proiezione di Futamura	78
4.9.1	Linguaggio Turing completo	79
5	Teoria della ricorsione	80
5.1	Insiemi ricorsivamente enumerabili	80
5.2	Insiemi ricorsivi	82

1 Introduzione

1.1 Cos'è l'informatica?

È una scienza che studia la calcolabilità, cioè cerca di capire che problemi si possono risolvere con un programma. Nasce dall'unione di matematica, ingegneria e logica. Il computer è solo uno strumento, mentre la matematica è il linguaggio con cui si creano algoritmi che permettono di risolvere i problemi.

1.2 Origini dell'informatica

Hilbert, nel 1900, si pose l'obiettivo di formalizzare tutta la matematica con un insieme finito e non contraddittorio di assiomi. Nel 1931, invece, Gödel dimostrò che l'informatica non potrà mai rappresentare tutta la matematica, perché ci saranno sempre proposizioni vere ma non dimostrabili tramite il calcolo. Ci si iniziò a chiedere se esistessero modelli di calcolo meccanici in grado di risolvere tutti i problemi. Nel 1936, Turing propose la macchina di Turing, una **sola** macchina programmabile in grado di risolvere tutti i problemi risolvibili:

$$\text{Int}(P, x) = \begin{cases} P(x) & \text{se } P(x) \text{ termina} \\ \uparrow & \text{se } P(x) \text{ non termina} \end{cases}$$

dove P è un programma e x è un input. La macchina di Turing è un modello teorico di calcolatore, che non esiste fisicamente, ma è in grado di simulare qualsiasi altro calcolatore. Da questo modello deriva la concezione di calcolabilità, cioè se un problema è intuitivamente calcolabile, allora esiste un programma in grado di risolverlo.

Altri modelli di calcolo che sono stati proposti sono:

- Lambda-calcolo
- Funzioni ricorsive
- Linguaggi di programmazione (Turing-completi)

Definizione utile 1.1. La Turing-completezza è la proprietà di un linguaggio di programmazione di essere in grado di simulare una macchina di Turing, cioè di poter risolvere qualsiasi problema risolvibile.

1.2.1 Calcolabilità

Un programma è calcolabile se termina, ma non è detto che termini in un tempo ragionevole. Non esistono algoritmi che possono dire se un programma termina o meno. Questo è un esempio di problema non calcolabile.

I problemi non calcolabili sono infinitamente più numerosi di quelli calcolabili

1.3 Nozioni di base

1.3.1 Basi di logica

Alcune nozioni di logica che ci serviranno in seguito:

- **Linguaggio del primo ordine:**

- Simboli relazionali (p, q, \dots)
- Simboli di funzione (f, g, \dots)
- Simboli di costante (c, d, \dots)

- **Simboli logici:**

- Parentesi (,) e virgola
- Insieme numerabile di variabili (v, x, \dots)
- Connettivi logici ($\neg, \wedge, \vee, \rightarrow, \leftrightarrow$)
- Quantificatori (\forall, \exists)

- **Termini:**

- Variabili
- Costanti
- f simbolo di funzione m -ario t_1, t_2, \dots, t_m termini, allora $f(t_1, t_2, \dots, t_m)$ è un termine.

- **Formula atomica:** p simbolo di relazione n -ario, t_1, t_2, \dots, t_n termini, allora $p(t_1, t_2, \dots, t_n)$ è una formula atomica.

- **Formula:**

- Formula atomica
- ϕ formula, allora $\neg\phi$ è una formula
- ϕ e ψ formule, allora $(\phi \wedge \psi), (\phi \vee \psi), (\phi \rightarrow \psi), (\phi \leftrightarrow \psi)$ sono formule.
- ϕ formula e v variabile, allora $\forall v.\phi$ e $\exists v.\phi$ sono formule.

1.3.2 Nozioni sugli insiemi

- $x \in A$ significa che x è un elemento dell'insieme A
- $\{x|P(x)\}$ si identifica insieme costituito dagli x che soddisfano la proprietà (o predicato) $P(x)$
- $A \subseteq B$ significa che A è un sottoinsieme di B se ogni elemento di A è anche in B
- $\mathcal{P}(S)$ denota l'insieme delle parti di S , ovvero l'insieme di tutti i sottoinsiemi di S ($\mathcal{P}(S) = \{X|X \subseteq S\}$)
- $A \setminus B = \{x|x \in A \wedge x \notin B\}, A \cup B = \{x|x \in A \vee x \in B\}, A \cap B = \{x|x \in A \wedge x \in B\}$
- $|A|$ denota la cardinalità di A , ovvero il numero di elementi in A .
- \bar{A} denota il complemento di A , ovvero $x \in \bar{A} \leftrightarrow x \notin A$

1.3.3 Nozioni sulle relazioni

- Prodotto cartesiano:

$$A_1 \times A_2 \times \cdots \times A_n = \{\langle a_1, a_2, \dots, a_n \rangle | a_1 \in A_1, \dots, a_n \in A_n\}$$

- Una **relazione** (binaria) è un sottoinsieme del prodotto cartesiano di (due) insiemi; dati A e B , $R \subseteq A \times B$ è una relazione su A e B
 - **Riflessiva**: $\forall a \in S$ si ha che aRa
 - **Simmetrica**: $\forall a, b \in S$ se aRb allora bRa
 - **Antisimmetrica**: $\forall a, b \in S$ se aRb e bRa allora $a = b$
 - **Transitiva**: $\forall a, b, c \in S$ se aRb e bRc allora aRc
- Per ogni relazione $R \subseteq S \times S$ la chiusura transitiva di R è il più piccolo insieme R^* tale che $\langle a, b \rangle \in R \wedge \langle b, c \rangle \in R \rightarrow \langle a, c \rangle \in R^*$
- Una relazione è detta **totale** su S se $\forall a, b \in S$ si ha che $aRb \vee bRa$
- Una relazione R di *di equivalenza* è una relazione binaria riflessiva, simmetrica e transitiva.
- Una relazione binaria $R \subseteq S \times S$ è un **pre-ordine** se è riflessiva e transitiva.
- R è un ordine parziale se è un pre-ordine antisimmetrico.
 - $x \in S$ è **minimale** rispetto a R se $\forall y \in S. y \not R x$ (ovvero $\neg(yRx)$)
 - $x \in S$ è **minimo** rispetto a R se $\forall y \in S. x R y$
 - $x \in S$ è **massimale** rispetto a R se $\forall y \in S. x \not R y$ (ovvero $\neg(xRy)$)
 - $x \in S$ è **massimo** rispetto a R se $\forall y \in S. y R x$

1.3.4 Nozioni sulle funzioni

- Una relazione f è una **funzione** se $\forall a \in A$ esiste uno ed un solo $b \in B$ tale che $(a, b) \in f$
- A dominio e B codominio di f . Il range di f è l'insieme di tutti i valori che f può assumere.
- f è **iniettiva** se $\forall a_1, a_2 \in A$ se $a_1 \neq a_2$ allora $f(a_1) \neq f(a_2)$
- Se $f : A \mapsto B$ è sia iniettiva che suriettiva allora è **bigettiva** e quindi esiste $f^{-1} : B \mapsto A$

1.4 Funzioni calcolabili

Un insieme è una proprietà ed è rappresentato da una funzione che indica se un elemento appartiene o meno all'insieme. I problemi da risolvere (in questo corso) hanno come soluzione una funzione sui naturali:

$$f : \mathbb{N} \rightarrow \mathbb{N}$$

Questo tipo di funzione è un **insieme** di associazioni input-output. Un esempio è la funzione quadrato:

$$f = \text{quadrato} = \{(0, 0), (1, 1), (2, 4), (3, 9), \dots\} = \{(n, n^2) | n \in \mathbb{N}\}$$

Quindi f è un insieme di coppie in $\mathbb{N} \subseteq \mathbb{N} \times \mathbb{N}$ la cui cardinalità è: $|\mathbb{N} \times \mathbb{N}| = |\mathbb{N}|$. Di conseguenza la funzione è un sottoinsieme di $\mathbb{N} \times \mathbb{N}$:

$$f \subseteq \mathbb{N} \times \mathbb{N} \quad f \in \mathcal{P}(\mathbb{N} \times \mathbb{N})$$

Dove $\mathcal{P}(\mathbb{N} \times \mathbb{N})$ è l'insieme delle parti di $\mathbb{N} \times \mathbb{N}$, cioè l'insieme di tutti i sottoinsiemi di $\mathbb{N} \times \mathbb{N}$.

Il numero di funzioni è:

$$f : \mathbb{N} \rightarrow \mathbb{N} = |\mathcal{P}(\mathbb{N} \times \mathbb{N})| = |\mathcal{P}(\mathbb{N})|$$

Esempio 1.1. Un esempio di insieme delle parti per l'insieme $A = \{1, 2, 3\}$ è:

$$\mathcal{P}(A) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$$

E le cardinalità sono:

$$|A| = 3 \quad |\mathcal{P}(A)| = 8 = 2^3$$



$$|\mathbb{N}| = \omega < \underbrace{|\mathcal{P}(\mathbb{N})| = 2^\omega}_{\substack{\text{Insieme delle funzioni,} \\ \text{non numerabile}}} = |\mathbb{R}|$$

Si ha quindi che **l'insieme delle funzioni non è numerabile**

Ci si chiede se queste funzioni sono tutte calcolabili:

Definizione 1.1. Una funzione **intuitivamente calcolabile** è una funzione descrivibile attraverso un algoritmo, cioè una sequenza finita di passi discreti elementari.

1.4.1 Quante funzioni numerabili ci sono?

Consideriamo Σ come un alfabeto finito, cioè una sequenza di simboli utilizzabili per scrivere un programma o algoritmo.

$$\Sigma = \{s_1, s_2, s_3, \dots, s_n\}$$

↓

Programma \subseteq sequenze di simboli in Σ

Con Σ^* si descrive l'insieme di tutte le sequenze finite di simboli in Σ , quindi l'insieme di tutti i possibili programmi è:

$$\text{Programmi} \subseteq \Sigma^*$$

Esempio 1.2. Se $\Sigma = \{a, b, c\}$ allora la sequenza di tutti i possibili simboli è:

$$\Sigma^* = \{\epsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, aab, \dots\}$$

dove ϵ è la stringa vuota.

La cardinalità di Σ^* è infinita numerabile (anche se Σ è finito).

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

Si ha quindi che l'insieme dei programmi è numerabile:

$$|\text{Programmi in } \Sigma| \leq |\Sigma^*| = |\mathbb{N}|$$

e questo implica che l'insieme delle **funzioni calcolabili è numerabile**

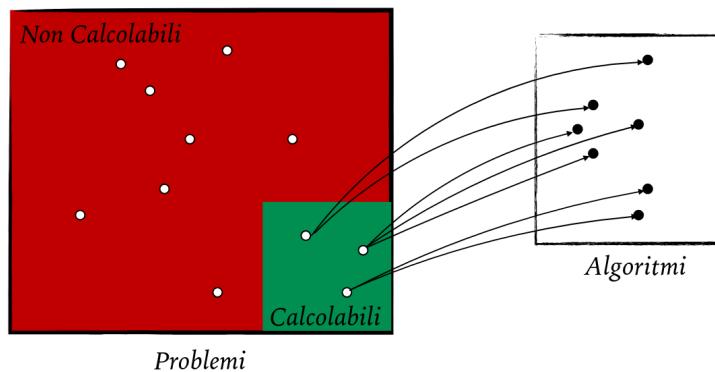


Figura 1: Cardinalità delle funzioni

Esempio 1.3. Prendiamo ad esempio la seguente funzione (serie di fibonacci):

$$f(n) \subseteq \mathbb{N} \times \mathbb{N}$$

Dove:

$$f(0) = 1, f(1) = 1, f(2) = 2$$

$$f(3) = 3, f(4) = 5, f(5) = 8$$

$$f(6) = 13, f(7) = 21, \dots$$

Definiamo un algoritmo ricorsivo:

$$\begin{cases} f(0) = 1 = f(1) \\ f(x+2) = f(x+1) + f(x) \end{cases}$$

Trovare un algoritmo non è possibile per tutte le funzioni, ma solo per quelle calcolabili.

Nell'insieme delle funzioni calcolabili ci sono:

- Funzioni totali, cioè definite per ogni input $n \in \mathbb{N}$ e terminano sempre
- Funzioni parziali, cioè non definite per ogni $n \in \mathbb{N}$

1.4.2 Funzioni vs Insiemi

Una funzione può essere vista come un linguaggio \mathcal{L}_f tale che:

$$f : \mathbb{N} \rightarrow \mathbb{N} \leftrightarrow \mathcal{L}_f = \left\{ 1^{f(x)} \mid x \in \mathbb{N} \right\} \quad \Sigma = \{1\}$$

Questo linguaggio permette di dire se un input appartiene o meno al linguaggio:

$$\begin{aligned} \sigma &\in \Sigma^* \\ &\downarrow \\ &\begin{cases} \sigma \in \mathcal{L}_f & \text{se appartiene al linguaggio} \\ \sigma \notin \mathcal{L}_f & \text{se non appartiene al linguaggio} \end{cases} \end{aligned}$$

Parliamo di insiemi invece che di funzioni dove gli elementi dell'insieme dipendono dal calcolo della funzione.

Esempio 1.4. Prendiamo ad esempio le seguenti funzioni:

- Funzione costante (Finite)

$$f(x) = 2 \rightarrow \mathcal{L}_f \text{ è finito}$$

- Funzione lineare (Regolari)

$$f(x) = 2x \rightarrow \mathcal{L}_f \text{ è infinito numerabile}$$

C'è bisogno di una memoria finita per determinare se la stringa appartiene al linguaggio

- (Context free)

$$f(\sigma) = \sigma \sigma^{\text{reverse}}$$

$$\sigma = abc \quad \sigma^{\text{reverse}} = cba$$

Per calcolare questa funzione c'è bisogno di una memoria illimitata, cioè non si può sapere a priori quanta ce n'è bisogno, è sufficiente uno stack.

- Decidibile

$$f(x) = x^2$$

Per calcolare questa funzione c'è bisogno di una memoria illimitata

Le funzioni calcolabili sono divise in classi secondo la gerarchia di Chomsky:

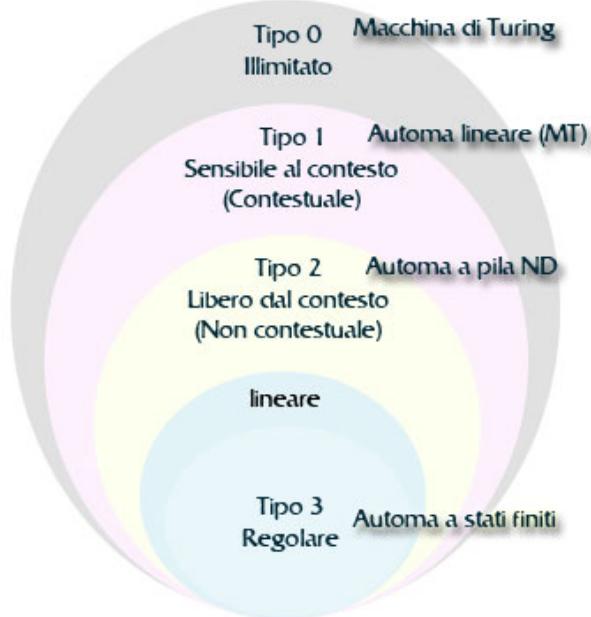


Figura 2: Gerarchia di Chomsky

2 Linguaggi regolari

Il principio di induzione è un meccanismo di definizione e dimostrazione che funziona **solo su insiemi infiniti**. Esistono due metodi di induzione:

- Induzione matematica
- Induzione strutturale

In questo corso tratteremo solo l'induzione matematica.

Un insieme A infinito con una relazione di ordine non riflessiva (senza l'uguale perché l'elemento non è in relazione con sé stesso): $< : (A, <)$. $A = \mathbb{N}$ e $<$ è l'ordinamento stretto tra numeri naturali. La relazione di ordine deve essere **ben fondata**, quindi non devono esserci catene discendenti infinite, cioè una sequenza di elementi in ordine decrescente infinita:

$$a_0 > a_1 > a_2 > a_3 > \dots \rightarrow \text{non ben fondata}$$

Una relazione di ordine riflessiva non è ben fondata, perché esistono catene infinite:

$$a_0 \geq a_1 \geq a_2 \geq a_3 \geq a_4 \geq \dots \rightarrow \text{non ben fondata}$$

b minimale in $A : b \in A$ b è minimale se $\forall b' < b . b' \notin A$ Ad esempio: $\{1, 2, 3\}$ ha come minimali (di contenimento) $\{1, 2\}$ e $\{2, 3\}$.

Definizione 2.1 (Principio di induzione). Se A è un insieme ben fondato (con ordinamento $<$), e Π è una proprietà definita sugli elementi di $A : \Pi \subseteq A$, allora:

$$\forall a \in A . \underbrace{\Pi(a)}_{a \text{ soddisfa } \Pi} \iff \underbrace{\forall a \in A . [\forall b < a . \Pi(b)] \Rightarrow \Pi(a)}_{\begin{array}{l} \text{Se dimostriamo } \Pi \text{ per ogni} \\ \text{elemento più piccolo di } a, \\ \text{allora } \Pi \text{ vale anche per } a \end{array}}$$

Consideriamo come caso base gli elementi minimali di A :

$$\text{Base}_A = \{a \in A \mid a \text{ minimale}\}$$

Se si dimostra che Π vale per tutti gli elementi minimali di A (la base):

$$\underbrace{\forall a \in \text{Base}_A . \Pi(a)}_{\begin{array}{l} \text{Base} \\ \text{Dimostriamo } \Pi \text{ per ogni} \\ \text{elemento minimale di } A \end{array}} \wedge \underbrace{\forall a \in A \setminus \text{Base}_A . \forall b < a . \Pi(b)}_{\begin{array}{l} \text{Passo induttivo} \\ \text{Ipotesi induttiva} \end{array}} \Rightarrow \Pi(a) \quad \text{Tesi da dimostrare}$$

Esempio 2.1. Dimostriamo che:

$$\forall n \in \mathbb{N} \sum_{i=1}^n i = \frac{n(n+1)}{2}$$

L'insieme è:

$$A = \mathbb{N} \setminus \{0\} = \{1, 2, 3, \dots\}$$

- **Base**

$$\text{Base}_A = \{1\}$$

– dimostriamo la base

$$\sum_{i=1}^1 i = n(n+1)/2 = 1(1+1)/2 = 1$$

- **Passo induttivo:** Prendo $n \in \mathbb{N}$

– Ipotesi induttiva, cioè per ogni $m < n$ vale la proprietà:

$$\forall m \in \mathbb{N} . \sum_{i=1}^m i = \frac{m(m+1)}{2}$$

Dobbiamo dimostrare la proprietà per n :

$$\sum_{i=1}^n i = \sum_{i=1}^{n-1} i + n$$

$n - 1 < n$ quindi vale l'ipotesi induttiva

$$\sum_{i=1}^{n-1} i = \frac{(n-1)(n-1+1)}{2} = \frac{(n-1)n}{2}$$

\downarrow

$$\begin{aligned} \sum_{i=1}^n i &= \sum_{i=1}^{n-1} i + n \\ &= \frac{(n-1)n}{2} + n \\ &= \frac{(n-1)n + 2n}{2} \\ &= \frac{n^2 - n + 2n}{2} \\ &= \frac{n(n+1)}{2} \quad \square \end{aligned}$$

È quindi dimostrato che:

$$\forall n \in \mathbb{N} \sum_{i=1}^n i = \frac{n(n+1)}{2}$$

2.1 Linguaggi formali

Definizione 2.2. Un linguaggio formale è un insieme di stringhe costruite su un alfabeto finito Σ .

Soltanamente un linguaggio formale \mathcal{L} è un sottoinsieme di Σ^* , tipicamente infiniti, ma non necessariamente:

$$\mathcal{L} \subseteq \Sigma^*$$

I linguaggi sono divisi in:

- Linguaggi finiti
- Linguaggi regolari, il modello utilizzato è l'automa a stati finiti.

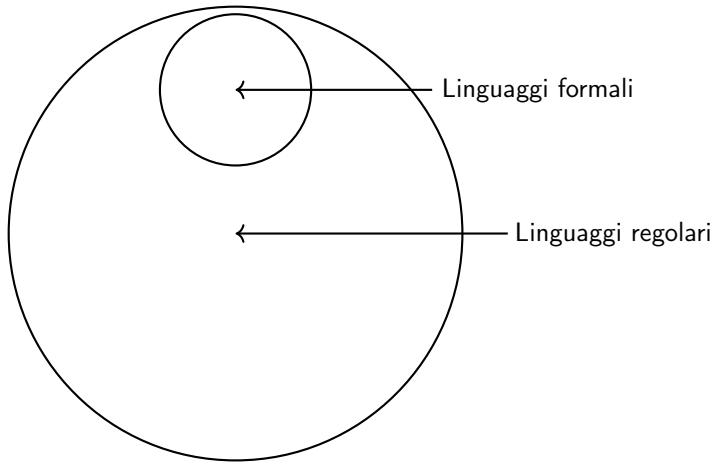


Figura 3: Linguaggi formali e linguaggi regolari

2.2 Linguaggi regolari (automi a stati finiti, DFA)

Il meccanismo più semplice per una memoria finita è l'automa a stati finiti

Consideriamo il linguaggio:

$$\mathcal{L}_f = \{1^{2n} \mid n \in \mathbb{N}\}$$

ha bisogno di due stati q_0 e q_1 . Lo stato q_0 rappresenta l'informazione di essere di lunghezza pari, mentre lo stato q_1 rappresenta l'informazione di essere di lunghezza dispari.

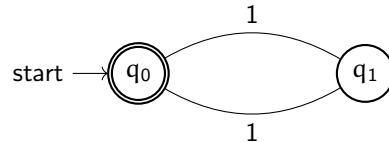


Figura 4: Automa a stati finiti per il linguaggio $\mathcal{L}_f = \{1^{2n} \mid n \in \mathbb{N}\}$

L'automa a stati finiti **deterministico** è definito come una quintupla:

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

dove:

- Q è un insieme **finito** di stati. Ogni stato rappresenta un'informazione
- Σ è un insieme **finito** di simboli (alfabeto). Ogni simbolo è un elemento atomico che posso leggere e che compone le stringhe da riconoscere
- $q_0 \in Q$ è uno stato e identifica lo stato iniziale. Lo stato finale viene indicato con un doppio cerchio
- $F \subseteq Q$ è l'insieme degli stati finali (di accettazione)

- $\delta : Q \times \Sigma \rightarrow Q$ È una **funzione di transizione** che dato uno stato e un simbolo, restituisce lo stato successivo ed è come se fosse una tabella che associa ad ogni coppia (stato, simbolo) uno stato:

$\Sigma \setminus Q$	q_0	q_1
1	q_1	q_0

La funzione deve essere **totale**, cioè deve essere definita per ogni coppia $(q, a) \in Q \times \Sigma$, quindi la tabella deve essere completa.

- $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$ Descrive lo stato che raggiungono leggendo una sequenza di simboli

$$\begin{cases} \hat{\delta}(q, \epsilon) = q \\ \hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a) \end{cases} \quad w \in \Sigma^*, \quad a \in \Sigma$$

È quindi la **chiusura transitiva** di δ

2.2.1 Come si dimostra che un linguaggio è regolare?

Esempio 2.2. Prendiamo in considerazione il seguente linguaggio:

$$L = \{\sigma \mid \sigma \text{ contiene almeno due } 1\}$$

$$\Sigma = \{0, 1\}$$

Con le seguenti stringhe si ha:

- $\sigma = 011 \in L$
- $\sigma = 1000100 \in L$
- $\sigma = 00010 \notin L$

L'informazione che codifica lo stato iniziale deve essere coerente con ϵ (stringa vuota)

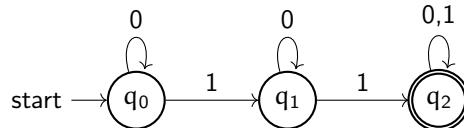


Figura 5: Automa a stati finiti per il linguaggio L

Un linguaggio L è riconosciuto da $M = (Q, \Sigma, \delta, q_0, F)$ (DFA, Deterministic Finite Automaton) se: $L = L(M)$ dove $L(M)$ è il linguaggio di M definito come:

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

Cioè sono tutte le stringhe che partendo da q_0 fanno raggiungere uno stato finale.

Definizione 2.3. Per dimostrare che L è regolare dobbiamo costruire M (almeno un M) e **dimostrare che** $L = L(M)$

$L = L(M)$ è un uguaglianza insiemistica e si dimostra con due contenimenti:

$$L = L(M) \equiv L \subset L(M) \wedge L(M) \subset L$$

- Se un elemento si trova nel primo insieme, allora si trova anche nel secondo

$$L \subseteq L(M) \equiv \sigma \in L \Rightarrow \sigma \in L(M) \equiv \sigma \in L \Rightarrow \hat{\delta}(q_0, \sigma) \in F$$

- Se un elemento si trova nel secondo insieme, allora si trova anche nel primo

$$L(M) \subseteq L \equiv \sigma \in L(M) \Rightarrow \sigma \in L \equiv \hat{\delta}(q_0, \sigma) \in F \Rightarrow \sigma \in L$$

o per contrapposizione:

$$\sigma \notin L \Rightarrow \hat{\delta}(q_0, \sigma) \notin F$$

Questo dimostra che il linguaggio è regolare perché è riconosciuto da un automa.

Esempio 2.3. Riprendendo l'esempio precedente:

$$L = \{\sigma \in \Sigma^* \mid \sigma \text{ contiene almeno due } 1\}$$

$$\Sigma = \{0, 1\}$$

$M =$

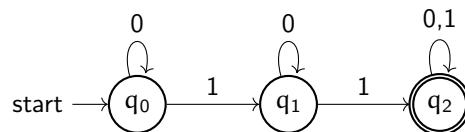


Figura 6: Automa a stati finiti per il linguaggio \mathcal{L}

Dimostriamo per induzione sulla lunghezza delle stringhe $\sigma \in \Sigma^*$ che se $x \in L$ allora $\hat{\delta}(q_0, x) \in F$ e se $x \notin L$ allora $\hat{\delta}(q_0, x) \notin F$.

$|\sigma| = 0$ non è **mai** sufficiente come base, ma è eventualmente la base **solo** per una delle due dimostrazioni. Bisogna quindi prendere la lunghezza più piccola che permette di avere sia $\sigma \in L$ che $\sigma \notin L$, in questo caso è $|\sigma| = 2$. Per ogni σ tale che $|\sigma| < 2 \quad \sigma \notin L$ perché non può contenere due 1 e non è riconosciuta da M dove il primo stato finale è raggiunto leggendo almeno due simboli.

$$\varepsilon \in L \quad \varepsilon \notin L$$

- **Base:** Controlliamo ogni stringa di lunghezza minima nel linguaggio

per provare il caso base. In questo caso la lunghezza minima è $|\sigma| = 2$

$$\begin{cases} \sigma = 11 \in L \text{ e } \hat{\delta}(q_0, 11) = q_2 \in F \\ \sigma = 10 \notin L \text{ e } \hat{\delta}(q_0, 10) = q_1 \notin F \\ \sigma = 01 \notin L \text{ e } \hat{\delta}(q_0, 01) = q_1 \notin F \\ \sigma = 00 \notin L \text{ e } \hat{\delta}(q_0, 00) = q_0 \notin F \end{cases}$$

- **Passo induttivo:** Assumiamo che valga l'**ipotesi induttiva**, cioè la tesi con un limite fissato:

$$\forall \sigma \in \Sigma^*. |\sigma| \leq n. \begin{cases} \sigma \in L \Rightarrow \hat{\delta}(q_0, \sigma) \in F \\ \sigma \notin L \Rightarrow \hat{\delta}(q_0, \sigma) \notin F \end{cases}$$

Vogliamo dimostrare che la tesi vale per $|\sigma| = n+1$ (la successiva stringa che posso considerare).

Tesi: $\begin{cases} \sigma \in L \Rightarrow \hat{\delta}(q_0, \sigma) = q_2 \text{ } \sigma \text{ contiene almeno due } 1 \\ \sigma \notin L \Rightarrow \hat{\delta}(q_0, \sigma) = q_0 \text{ } \sigma \text{ non contiene } 1 \\ \sigma \notin L \Rightarrow \hat{\delta}(q_0, \sigma) = q_1 \text{ } \sigma \text{ contiene esattamente un } 1 \end{cases}$

Ipotesi induttiva:

$$\forall \sigma \in \Sigma^*. |\sigma| \leq n. \text{ allora la tesi vale su } \sigma$$

Dimostrazione della tesi per σ tale che $|\sigma| = n + 1$. ($|\sigma'| = n$ quindi su σ' possiamo applicare l'ipotesi induttiva):

$$|\sigma| = n + 1 \rightarrow \sigma = \sigma'1 \vee \sigma = \sigma'0$$

- Supponiamo che σ appartenga al linguaggio e termini con 1:

$$\sigma \in L \wedge \sigma = \sigma'1$$

↓

* Se $\sigma' \in L$ applico l'ipotesi induttiva:

$$\hat{\delta}(q_0, \sigma') = q_2$$

$$\begin{aligned} \hat{\delta}(q_0, \sigma) &\stackrel{\sigma=\sigma'1}{=} \hat{\delta}(q_0, \sigma'1) \\ &= \delta(\hat{\delta}(q_0, \sigma'), 1) \\ &= \delta(q_2, 1) = q_2 \in F \end{aligned}$$

* Se $\sigma' \notin L$ allora σ' contiene esattamente un 1:

$$\hat{\delta}(q_0, \sigma') = q_1$$

$$\hat{\delta}(q_0, \sigma'1) = \delta(q_1, 1) = q_2$$

- Supponiamo che σ appartenga al linguaggio e termini con 0:

$$\sigma \in L \wedge \sigma = \sigma'0$$

Per definizione di L abbiamo che

$$\sigma \in L \wedge \sigma = \sigma'0 \Rightarrow \sigma' \in L$$

Dimostriamo l'ipotesi induttiva:

$$\hat{\delta}(q_0, \sigma') = q_2$$

allora

$$\begin{aligned} \hat{\delta}(q_0, \sigma) &= \hat{\delta}(q_0, \sigma'0) \\ &= \delta(\hat{\delta}(q_0, \sigma'), 0) \\ &= \delta(q_2, 0) = q_2 \in F \end{aligned}$$

- Supponiamo che σ non appartenga al linguaggio e contiene esattamente un 1:

* $\sigma = \sigma'0 \Rightarrow \sigma' \notin L$ e contiene esattamente un 1
 Ipotesi induttiva:

$$\hat{\delta}(q_0, \sigma') = q_1$$

\Downarrow

$$\begin{aligned} \hat{\delta}(q_0, \sigma) &= \hat{\delta}(q_0, \sigma'0) \\ &= \delta(\hat{\delta}(q_0, \sigma'), 0) \\ &= \delta(q_1, 0) = q_1 \notin F \end{aligned}$$

* $\sigma = \sigma'1 \Rightarrow \sigma' \notin L$ e non contiene 1
 Ipotesi induttiva:

$$\hat{\delta}(q_0, \sigma') = q_0$$

\Downarrow

$$\begin{aligned} \hat{\delta}(q_0, \sigma) &= \hat{\delta}(q_0, \sigma'1) \\ &= \delta(\hat{\delta}(q_0, \sigma'), 1) \\ &= \delta(q_0, 1) = q_1 \notin F \end{aligned}$$

- Supponiamo che σ non appartenga al linguaggio e non contiene 1

$$\sigma \notin L \wedge \sigma = \sigma'0$$

$\sigma = \sigma'1$ non è possibile per l'ipotesi che σ non contiene 1

Ipotesi induttiva:

$$\hat{\delta}(q_0, \sigma') = q_0$$

\Downarrow

$$\begin{aligned} \hat{\delta}(q_0, \sigma) &= \hat{\delta}(q_0, \sigma'0) \\ &= \delta(\hat{\delta}(q_0, \sigma'), 0) \\ &= \delta(q_0, 0) = q_0 \notin F \end{aligned}$$

Tutti i casi sono dimostrati, quindi abbiamo dimostrato che:

$$L = L(M) \Rightarrow L \text{ è regolare}$$

Esercizio 2.1. Consideriamo il seguente linguaggio:

$$L = \{\sigma \in \Sigma^* \mid \text{ogni sequenza di } 0 \text{ è di lunghezza pari}\}$$

$$\Sigma = \{0, 1\}$$

Si può accettare anche sequenze di lunghezza 0. Alcuni esempi sono:

$$101 \notin L$$

$$1111 \in L$$

$$10010000 \in L$$

$$00101 \notin L$$

L'automa a stati finiti M è il seguente:

- q_0 : Non sono stati letti 0
- q_1 : Sequenza di 0 consecutiva di lunghezza dispari
- q_2 : Sequenza di 0 consecutiva di lunghezza pari

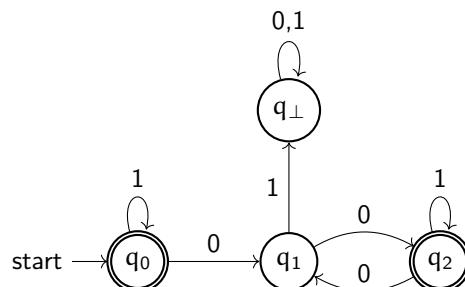


Figura 7: Automa a stati finiti per il linguaggio L

La tesi è:

- $\sigma \in L$ e non contiene 0: $\hat{\delta}(q_0, \sigma) = q_0$
- $\sigma \in L$ e contiene una sequenza pari di 0: $\hat{\delta}(q_0, \sigma) = q_2$
- $\sigma \notin L$ e contiene una sequenza **finale** dispari di 0: $\hat{\delta}(q_0, \sigma) = q_1$
- $\sigma \notin L$ e contiene una sequenza dispari di 0 seguita da 1: $\hat{\delta}(q_0, \sigma) = q_{\perp}$

Esercizio 2.2. Consideriamo il seguente linguaggio (ogni sequenza di 0 è di lunghezza almeno 2):

$$L = \{\sigma \in \Sigma^* \mid \exists n \geq 1 . \sigma = 0^n \Rightarrow n \geq 2\}$$

$$\Sigma = \{0, 1\}$$

L'automa a stati finiti M è il seguente:

- q_0 : Non contiene 0, oppure **tutte** le sequenze di 0 sono lunghe almeno 2
- q_1 : Esattamente uno 0
- q_2 : Almeno due 0

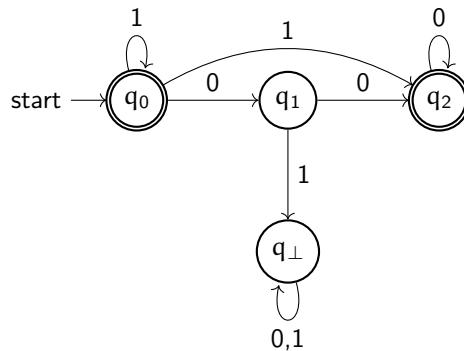


Figura 8: Automa a stati finiti per il linguaggio L

La tesi è:

- $\sigma \in L$ e $\sigma = \sigma'1 \Rightarrow \hat{\delta}(q_0, \sigma) = q_0$
- $\sigma \in L$ e $\sigma = \sigma'0 \Rightarrow \hat{\delta}(q_0, \sigma) = q_2$
- $\sigma \notin L$ e $\sigma = \sigma'0$ dove l'ultima sequenza di 0 è esattamente lunga 1:
 $\Rightarrow \hat{\delta}(q_0, \sigma) = q_1$
- $\sigma \notin L$ e σ contiene una sequenza lunga 1 di 0: $\Rightarrow \hat{\delta}(q_0, \sigma) = q_{\perp}$

2.3 Automi a stati finiti non deterministici (NFA)

Un automa a stati finiti non deterministico si crea quando ad un solo simbolo sono associate più transizioni. Quando questo succede gli stati vengono considerati in parallelo. Un NFA è definito come una quintupla:

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

- Q è un insieme finito di stati
- Σ è un insieme finito di simboli (alfabeto)

- $q_0 \in Q$ è uno stato e identifica lo stato iniziale
- $F \subseteq Q$ è l'insieme degli stati finali
- $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$ è una funzione di transizione che dato uno stato e un simbolo restituisce un insieme di stati **potenzialmente** raggiungibili. È possibile che esistano coppie associate all'insieme vuoto:

$$\emptyset \in \mathcal{P}(Q)$$

Inoltre non è obbligatorio avere un arco uscente per ogni simbolo di Σ .

- $\hat{\delta} : Q \times \Sigma^* \rightarrow \mathcal{P}(Q)$ Descrive gli stati che si possono raggiungere leggendo una sequenza di simboli:

$$\begin{cases} \hat{\delta}(q, \epsilon) = \{q\} \\ \hat{\delta}(q, wa) = \bigcup_{p \in \hat{\delta}(q, w)} \delta(p, a) \end{cases} \quad w \in \Sigma^*, \quad a \in \Sigma$$

È quindi la chiusura transitiva di δ

Esempio 2.4. Un esempio di NFA è il seguente:

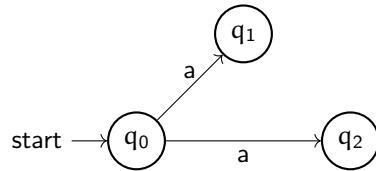


Figura 9: Esempio di NFA

$$\delta(q_0, a) = \{q_1, q_2\} \subseteq Q$$

2.3.1 Linguaggio riconosciuto da un NFA

Un linguaggio L è riconosciuto da un NFA N se:

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

Esempio 2.5. Consideriamo il seguente linguaggio:

$$L(N) = \{ \sigma \in \Sigma^* \mid \sigma \text{ contiene almeno due } 1 \}$$

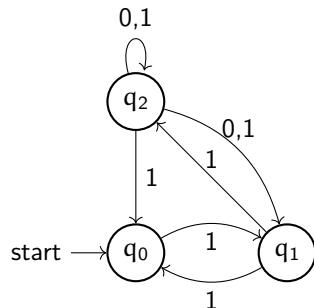


Figura 10: Esempio di NFA per il linguaggio L

Teorema 2.1 (Teorema di Rabin-Scott). Ogni linguaggio riconosciuto da un NFA è riconosciuto da un DFA.

$$\forall N = (Q, \Sigma, \delta, q_0, F) \exists M = (Q', \Sigma, \delta', q'_0, F') . L(N) = L(M)$$

Dimostrazione: Consideriamo un NFA $N = (Q, \Sigma, \delta, q_0, F)$ e costruiamo un DFA $M = (Q', \Sigma, \delta', q'_0, F')$. Gli insiemi degli stati sono:

$$Q' = \mathcal{P}(Q)$$

$$Q = \{q_0, q_1, q_2\}$$

↓

$$Q' = \left\{ \emptyset, \{q_0\}, \{q_1\}, \{q_2\}, \{q'_3\}, \{q'_4\}, \{q'_5\}, \{q'_6\}, \{q'_7\}, \{q_0, q_1\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\} \right\}$$

Lo stato iniziale rimane uguale per entrambi gli insiemi: $q'_0 = \{q_0\}$.

Gli insiemi degli stati finali sono:

$$F' = \{P \subseteq Q \mid P \cap F \neq \emptyset\} \quad P \in \mathcal{P}(Q)$$

Quindi:

$$F = \{q_2\}$$

$$F' = \left\{ \{q'_3\}, \{q'_6\}, \{q'_5\}, \{q'_7\}, \{q_2\}, \{q_1, q_2\}, \{q_0, q_2\}, \{q_0, q_1, q_2\} \right\}$$

La funzione di transizione è definita come:

$$\delta'(P, a) = \bigcup_{q \in P} \delta(q, a) \in \mathcal{P}(Q) \quad P \in Q' = \mathcal{P}(Q), \quad a \in \Sigma$$

Quindi:

$$\begin{aligned}\underbrace{\delta'(q'_5, 1)}_{=\{q_1, q_2\}} &= \delta(q_1, 1) \cup \delta(q_2, 1) \\ &= \{q_0, q_2\} \cup \{q_0, q_1, q_2\} \\ &= \{q_0, q_1, q_2\} = q'_7\end{aligned}$$

Dimostriamo:

$$1. \hat{\delta}(q_0, \sigma) = \hat{\delta}'(q'_0, \sigma) = \{q_0\}$$

Dimostriamo per induzione su $|\sigma|$:

- Se $\sigma = \varepsilon$ allora:

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

per le definizioni

- Se $\sigma = \sigma' a$:

$$\begin{aligned}\hat{\delta}'(q'_0, \sigma' a) &= \delta'(\hat{\delta}'(q_0, \sigma'), a) \\ &= \delta'(\hat{\delta}(q_0, \sigma'), a) \\ &= \bigcup_{p \in \hat{\delta}(q_0, \sigma')} \delta(p, a) \\ &= \hat{\delta}(q_0, \sigma' a)\end{aligned}$$

Definizione di $\hat{\delta}'$ non deterministica

$$2. \sigma \in L(N) \iff \sigma \in L(M)$$

Dimostriamo la definizione di linguaggio riconosciuto in NFA:

$$\begin{aligned}\sigma \in L(N) &\iff \hat{\delta}(q_0, \sigma) \cap F \neq \emptyset \\ &\iff \hat{\delta}'(q'_0, \sigma) \cap F' \neq \emptyset ((1.)) \\ &\iff \hat{\delta}'(q'_0, \sigma) \in F' \\ &\iff \sigma \in L(M) (\text{def. linguaggio accettato in DFA})\end{aligned}$$

2.3.2 Conversione da NFA a DFA

Prendiamo in considerazione il seguente NFA:

$$L(N) = \{\sigma \in \Sigma^* \mid \sigma \text{ contiene almeno due } 1\}$$

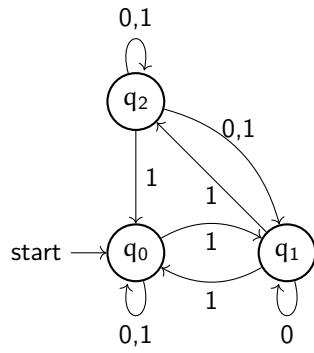


Figura 11: Esempio di NFA per il linguaggio L

Per trasformare un NFA in un DFA bisogna creare dei nuovi stati che raggruppano gli stati non deterministici. In questo caso:

- **Tabella degli stati della NFA:**

Stato	Input 0	Input 1
q_0	$\{q_0\}$	$\{q_0, q_1\}$
q_1	$\{q_1\}$	$\{q_0, q_2\}$
q_2	$\{q_1, q_2\}$	$\{q_0, q_1, q_2\}$

Tabella 1: Tabella di transizione della NFA

- **Traduzione degli stati della NFA in stati del DFA:**

		0	1
	\emptyset	\emptyset	\emptyset

Tabella 2: Tabella di traduzione degli stati della NFA in stati del DFA

2.4 Automi non deterministici con ε -transizioni (ε -NFA)

Questo tipo di NFA permette di cambiare stato anche senza leggere simboli:

$$q_1 \xrightarrow{\varepsilon} q_2$$

Un ε -NFA è definito come un NFA con la differenza che la funzione di transizione è definita come:

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow \underbrace{\mathcal{P}(Q)}_{\text{Non determinismo}}$$

Esempio 2.6. Prendiamo ad esempio il seguente ε -NFA in cui leggendo solo a si può raggiungere sia q' che q'' :

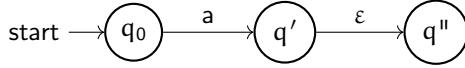


Figura 12: Esempio di ε -NFA

2.4.1 ε -closure

Per definire $\hat{\delta}$ bisogna prima definire la ε -closure.

Una ε -closure di uno stato q è l'insieme di tutti gli stati che si possono raggiungere da q seguendo archi etichettati con ε :

$$\varepsilon\text{-closure} : Q \rightarrow \mathcal{P}(Q)$$

O definito per insiemi:

$$\varepsilon\text{-closure} : \mathcal{P}(Q) \rightarrow \mathcal{P}(Q)$$

$$\varepsilon\text{-closure}(P) = \bigcup_{p \in P} \varepsilon\text{-closure}(p)$$

La funzione $\hat{\delta}$ è definita come:

$$\begin{cases} \hat{\delta}(q, \varepsilon) &= \varepsilon\text{-closure}(q) \\ \hat{\delta}(q, wa) &= \bigcup_{p \in \hat{\delta}(q, w)} \varepsilon\text{-closure}(\delta(p, a)) \end{cases}$$

Esempio 2.7. La ε -closure dell'esempio precedente è:

$$\varepsilon\text{-closure}(q') = \{q', q''\}$$

Il riconoscimento di un linguaggio è analogo a quello di un NFA:

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

Teorema 2.2. Sia $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ una ε -NFA, allora esiste una NFA M' tale che $L(M) = L(M')$.

Quindi l'insieme dei linguaggi riconosciuti da ε -NFA coincide con quello degli NFA, che a sua volta coincide con i linguaggi regolari.

$$M = \langle Q, \Sigma, \delta, q_0, F \rangle \text{ } \varepsilon\text{-NFA}$$

Costruiamo una NFA

$$M' = \langle Q', \Sigma', \delta', q'_0, F' \rangle$$

Dove:

$$Q' = Q, \quad \Sigma' = \Sigma, \quad q'_0 = q_0$$

$$\delta'(q, a) = \hat{\delta}(q, a) \quad F' = \begin{cases} F \cup \{q_0\} \text{ se } \varepsilon\text{-closure}(q_0) \cap F \neq \emptyset \\ F \end{cases}$$

in cui il numero degli stati rimane lo stesso, l'alfabeto non cambia e neanche lo stato iniziale.

2.5 Espressioni regolari

Le espressioni regolari sono operazioni algebriche sui linguaggi regolari. Le operazioni che si possono fare sono:

- **Unione:** Dati i linguaggi $L_1, L_2 \subseteq \Sigma^*$

$$L_1 \cup L_2 = \{\sigma \mid \sigma \in L_1 \vee \sigma \in L_2\}$$

- **Concatenazione:** Dati i linguaggi $L_1, L_2 \subseteq \Sigma^*$

$$L_1 \cdot L_2 = \{\sigma_1 \sigma_2 \mid \sigma_1 \in L_1 \wedge \sigma_2 \in L_2\} \equiv L_1 L_2$$

Ad esempio:

$$L_1 = \{0101, 010101\} \quad L_2 = \{000, 111\}$$

$$L_1 \cdot L_2 = \{0101000, 0101111, 010101000, 010101111\}$$

- **Stella di Kleene:** Dato un linguaggio $L \subseteq \Sigma^*$

$$L^* = \bigcup_{n \in \mathbb{N}} L^n$$

Cioè la concatenazione di L con se stesso n volte, con:

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

Ad esempio:

$$L = \{000, 111\}$$

$$L^0 = \{\varepsilon\}$$

$$L^1 = L \cdot L^0 = \{000, 111\}$$

$$L^2 = L \cdot L^1 = \{000000, 000111, 111000, 111111\}$$

$$L^3 = L \cdot L^2 = \left\{ \begin{array}{l} 000000000, 000000111, 000111000, 000111111, \\ 111000000, 111000111, 111111000, 111111111 \end{array} \right\}$$

⋮

$$L^* = \{L^0, L^1, L^2, L^3, \dots\}$$

- L^* è l'insieme di tutte le possibili concatenazioni di stringhe appartenenti a L , compresa la stringa vuota ε
- $L^+ = \bigcup_{n \geq 0} L^n$ è definito come L^* senza la stringa vuota ε :

$$L^+ = L \cdot L^*$$

Definizione 2.4. Definiamo per induzione le espressioni regolari su un alfabeto Σ :

- **Caso base:**

- $\emptyset \subseteq \Sigma^*$ è un'espressione regolare che rappresenta il linguaggio vuoto
- ε è un'espressione regolare che rappresenta il linguaggio:

$$\{\varepsilon\} \subseteq \Sigma^*$$

- $a \in \Sigma$ è un'espressione regolare che rappresenta il linguaggio:

$$\{a\} \subseteq \Sigma^*$$

- **Passo induttivo:**

Siamo r, s sono espressioni regolari che rappresentano il linguaggio

$$R \subseteq \Sigma^* \wedge S \subseteq \Sigma^*$$

- $r + s$ è un'espressione regolare che rappresenta il linguaggio $R \cup S$
- $r \cdot s$ è un'espressione regolare che rappresenta il linguaggio $R \cdot S$
- r^* è un'espressione regolare che rappresenta il linguaggio:

$$R^*$$

Esempio 2.8. Prendiamo ad esempio l'espressione regolare:

$$1^* + 0^* + (10)^*$$

che equivale a

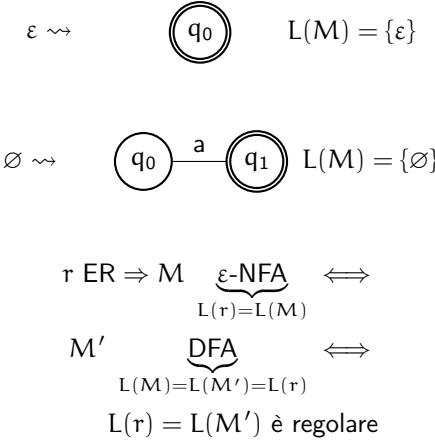
$$\{1^n \mid n \in \mathbb{N}\} \cup \{0^n \mid n \in \mathbb{N}\} \cup \{(10)^n \mid n \in \mathbb{N}\}$$

Teorema 2.3 (Teorema di equivalenza). Dato un DFA $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ allora esiste un'espressione regolare r tale che $L(M) = L(r)$.

$$L \text{ Regolare} \stackrel{\text{def}}{\iff} \underbrace{\exists M \text{ DFA} . L}_{L(M)=L} \stackrel{\text{Th}}{\Rightarrow} \exists r \in \underbrace{\text{ER}}_{\text{Espressioni Regolari}} . L(r) = L$$

Teorema 2.4. Data un'espressione regolare (ER) r esiste una ε -NFA M tale che: $L(r) = L(M)$

Quindi:



Esempio 2.9. Consideriamo il seguente linguaggio:

$$L = \{\sigma \in \Sigma^* \mid \sigma \text{ contiene almeno due } 1\}$$

Per dimostrare che è regolare si costruisce il DFA M e si dimostra $L = L(M)$. Però se si ha un'espressione regolare $r = 0^* 1 0^* 1 0^*$ **non** dimostra che L è regolare. Bisognerebbe dimostrare $L = L(r)$

2.6 Proprietà dei linguaggi regolari

2.6.1 Proprietà di chiusura

Indica se l'insieme dei linguaggi regolari è chiuso rispetto ad alcune operazioni, cioè se applicando queste operazioni a linguaggi regolari si ottengono sempre linguaggi regolari.

Operazioni:

- *
- \cup
- \cdot
- \cap ($L_1 \cap L_2 = \{\sigma \mid \sigma \in L_1 \wedge \sigma \in L_2\}$)
- $\bar{\cdot}$ ($\bar{L} = \{\sigma \mid \sigma \notin L\}$)

Teorema 2.5. I linguaggi regolari sono chiusi rispetto alle operazioni di:

- Stella di Kleene
- Unione (finita)

- Concatenazione

Consideriamo i linguaggi regolari L_1, L_2 . Allora:

- L_1^* è regolare
- $L_1 \cup L_2$ è regolare
- $L_1 \cdot L_2$ è regolare

Teorema 2.6. I linguaggi regolari sono chiusi rispetto alla complementazione. Dato un automa a stati finiti, il linguaggio complementare è riconosciuto dall'automa complementare, cioè quello in cui gli stati finali diventano non finali e viceversa.

Per le leggi di De Morgan, i linguaggi regolari sono chiusi per intersezione finita:

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

2.6.2 Proprietà di decidibilità

Indica se esistono algoritmi che risolvono alcuni problemi sui linguaggi regolari. Consideriamo un insieme di stringhe accettate da un DFA (linguaggio regolare) M con n stati.

- $L(M) \neq \emptyset$ se e solo se accetta almeno una stringa di lunghezza $\leq n$
- $L(M)$ è infinito se e solo se accetta almeno una stringa di lunghezza l con $n \leq l < 2n$. Cioè se esiste un ciclo. Questo fornisce un estremo superiore per verificare se un linguaggio è infinito.
- $L(M_1) = L(M_2)$

2.6.3 Esistenza dell'automa minimo

Forniamo strategie per costruire un automa minimo.

Definizione 2.5 (Relazione di equivalenza e partizione). Data una relazione R su Σ . R è una relazione di:

- Equivalenza: $E \subset \Sigma \times \Sigma$
- Riflessiva: $\forall a . aRa$
- Simmetrica: $\forall a, b . aRb \Rightarrow bRa$
- Transitiva: $\forall a, b, c . aRb \wedge bRc \Rightarrow aRc$

La **relazione di equivalenza R induce una partizione** (unione di insiemi) di S :

$$R \subseteq S \times S$$

ovvero:

$$S = S_1 \cup S_2 \cup \dots \cup S_k$$

dove S_i sono una partizione di S . Inoltre:

- $\forall i, j . S_i \cap S_j = \emptyset$
- $\forall i . \forall a, b \in S_i . a R b$
- $\forall a \in S_i, b \in S_j, i \neq j . a \not R b$

Quindi R è come se dividesse S in insiemi disgiunti:

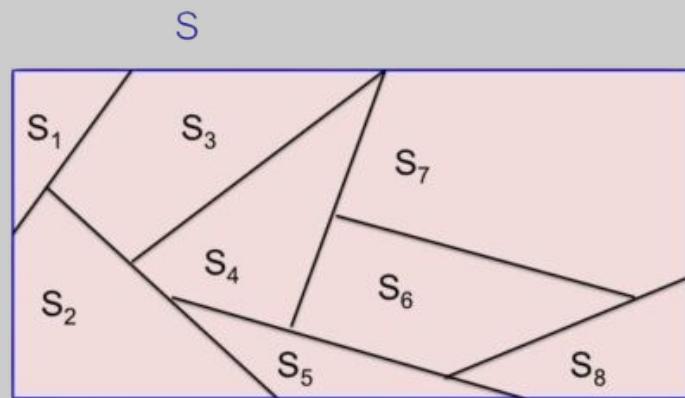


Figura 13: Esempio di partizione su S

S_i sono detti **classi di equivalenza** di R e si indica con:

$$a \in R \Rightarrow [a]_R = \{b \mid a R b\}$$

Ad esempio:

$$c R a \Rightarrow [a]_R \equiv [c]_R$$

Definiamo di seguito due relazioni di equivalenza:

Definizione 2.6 (Classe di equivalenza definita per tutti i linguaggi). Consideriamo un linguaggio $L \subseteq \Sigma^*$, possiamo definire una relazione R_L come:

$$R_L \subseteq \Sigma^* \times \Sigma^*$$

$$x, y \in \Sigma^* \quad x R_L y \iff \forall z \in \Sigma^* . xz \in L \iff yz \in L$$

Quindi o entrambi appartengono a L o entrambi non appartengono a L .

Definizione 2.7 (Classe di equivalenza definita per gli automi). Consideriamo un automa DFA $\langle Q, \Sigma, \delta, q_0, F \rangle$. Possiamo definire una relazione R_M come:

$$R_M \subseteq \Sigma^* \times \Sigma^*$$

$$x, y \in \Sigma^* \quad x R_M y \iff \delta(q_0, x) = \delta(q_0, y)$$

Due stringhe sono in relazione se raggiungono lo stesso stato.

Definizione 2.8 (Relazione invariante destra). Una relazione R su Σ^* : $R \subseteq \Sigma^* \times \Sigma^*$ è **invariante destra** se e solo se:

$$x, y \in \Sigma^* \quad x R y \implies \forall z \in \Sigma^*. xz R yz$$

Se due stringhe sono in relazione tra di loro, allora in qualunque modo vengano estese, rimarranno in relazione tra di loro. Quindi essere in relazione è invariante rispetto all'estensione della stringa **verso destra**

R_L e R_M sono relazioni invarianti destre.

Definizione 2.9 (Raffinamento). La relazione R_2 è **raffinamento** di R_1 se:

$$\begin{aligned} R_2 &\subseteq S \times S \\ R_1 &\subseteq S \times S \quad \text{di equivalenza} \end{aligned}$$

e R_1 è più grossa di R_2 , cioè ogni classe di equivalenza di R_2 è contenuta in una classe di equivalenza di R_1 e quindi è come se fosse più dettagliata:

$$\forall x. [x]_{R_2} \subseteq [x]_{R_1}$$

Il numero di classi di equivalenza di R_2 è maggiore del numero di classi di equivalenza di R_1 .

Definizione 2.10. Dato un automa DFA $M = \langle Q, \Sigma, \delta, q_0, F \rangle$, ogni stato definisce un linguaggio L_q come:

$$q \in Q : L_q = \{ \sigma \in \Sigma^* \mid \hat{\delta}(q, \sigma) = q \}$$

Teorema 2.7 (Teorema di Myhill-Nerode). I seguenti enunciati sono equivalenti:

1. $L \subseteq \Sigma^*$ è un linguaggio regolare, ovvero esiste un DFA M tale che

$$L = L(M)$$

2. L è unione di classi di equivalenza (cioè è partizionato) indotte da una relazione di equivalenza R invariante destra e di indice finito, cioè se il numero di classi di equivalenza indotte è finito.
3. R_L è di indice finito

Dimostriamo che:

$$1. \implies 2. \underset{R \text{ raffina } R_L}{\implies} 3. \underset{\text{Costruisce } M}{\implies} 1.$$

- 1. \implies 2.

Ipotesi: L è un linguaggio riconosciuto da un DFA $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ $L = L(M)$.

Tesi: Esistete una relazione di equivalenza R invariante destra e di indice finito tale che L è unione di classi di equivalenza di R .

Prendiamo $R = R_M$ $xR_My \iff \delta(q_0, x) = \delta(q_0, y)$.

1. Il numero delle classi di equivalenza di R_M è uguale al numero di stati di M : $|Q|$ e $|Q|$ è finito, quindi R_M è di indice finito
2. R_M è invariante destra

$$\begin{aligned} L = L(M) &= \{x \in \Sigma^* \mid \hat{\delta}(q_0, x) \in F\} \\ &= \{x \in \Sigma^* \mid \hat{\delta}(q_0, x) = q \wedge q \in F\} \\ &= \bigcup_{q \in F} \{x \in \Sigma^* \mid \hat{\delta}(q_0, x) = q\} \\ &= \bigcup_{q \in F} L_q \end{aligned}$$

Quindi L è unione di classi di equivalenza di R_M .

- 2. \implies 3.

Ipotesi: L è unione di classi di equivalenza di una relazione di equivalenza R invariante destra e di indice finito.

Tesi: R_L è di indice finito (numero di classi finito).

Dimostriamo che R è raffinamento di R_L perché allora il numero di classi di equivalenza di R (finito per ipotesi) sarebbe maggiore del numero di classi di R_L (che quindi sarebbe finito).

Per dimostrare R raffinamento di R_L bisogna dobbiamo dimostrare che se due oggetti sono in relazione secondo la relazione più fine R , lo sono

anche secondo la relazione più grossa R_L :

$$\begin{aligned} \forall x, y . xRy &\implies xR_L y \\ &\equiv [y \in [x]_R \implies y \in [x]_{R_L}] \\ &\equiv [x]_R \subseteq [x]_{R_L} \text{ per raffinamento} \end{aligned}$$

Prendiamo xRy sapendo che R è invariante destra, cioè:

$$\forall z \in \Sigma^* . xRy \implies xzRyz$$

$L = \cup$ classi di equivalenza

Questo implica che o entrambe x e y appartengono a L o entrambe non appartengono a L :

$$xRy \implies x \in L \iff y \in L$$

$$[x]_R \subseteq L \vee [x]_R \text{ fuori da } L$$

$$\begin{aligned} xRy &\implies x \in L \iff y \in L \\ &\stackrel{\substack{\implies \forall z . xzRyz \\ \text{Invariante destra}}}{\implies} \stackrel{\substack{\implies xz \in L \\ L = \cup \text{ classi}}}{\implies} yz \in L \\ &\stackrel{\substack{\implies xR_L y \\ \text{Def. di } R_L}}{\implies} \end{aligned}$$

Questo dimostra che dato un M generico R_M è un raffinamento di R_L .

- 3. \implies 1. **Ipotesi:** R_L è di indice finito.

Tesi: L è regolare, cioè è riconosciuto da un DFA M : $L = L(M)$.

Costruiamo M :

- $Q = \{[x]_{R_L} \mid x \in \Sigma^*\}$ è l'insieme delle classi di equivalenza di R_L (sono finite per ipotesi)
- Σ è l'alfabeto del linguaggio L
- $q_0 = [\varepsilon]_{R_L}$ è la classe di equivalenza della stringa vuota
- $F = \{[x]_{R_L} \mid x \in L\}$ è l'insieme delle classi di equivalenza che contengono almeno una stringa appartenente a L
- $\delta(q, a) = \delta([x]_{R_L}, a) = [xa]_{R_L}$. Si potrebbe anche prendere un elemento qualsiasi $y \in [x]_{R_L}$ e definire:

$$yRx \implies [x]_{R_L} = [y]_{R_L} \implies [ya]_{R_L} = [xa]_{R_L}$$

e questo vale perché R_L è invariante destra. Quindi la definizione di δ è una buona definizione perché è indipendente dall'elemento che rappresenta la classe di equivalenza.

Bisogna dimostrare che $L = L(M)$:

– Dimostrazione per induzione che $\hat{\delta}([x], y) = [xy]$

$$\begin{aligned}\hat{\delta}(q_0, x) &= \hat{\delta}([\varepsilon]_{R_L}, x) = [x]_{R_L} \\ \implies x \in L(M) &\iff \hat{\delta}(q_0, x) \in F \\ &\iff \hat{\delta}([\varepsilon]_{R_L}, x) \in F \\ &\iff [x]_{R_L} \in F \\ &\iff x \in L \implies L(M) = L\end{aligned}$$

Dato L esiste un DFA M tale che $L = L(M)$ ed M ha il numero minimo di stati. (Quello costruito con le classi di equivalenza di R_L)

2.6.4 Condizione necessaria perché un linguaggio sia regolare

Un linguaggio L è regolare se esiste un automa, quindi per dimostrare che un linguaggio non è regolare si può dimostrare che non esiste un automa. Per fare ciò si usa il Pumping lemma che fornisce una condizione Π_L **necessaria** alla regolarità di un linguaggio:

$$L \text{ regolare} \Rightarrow \Pi_L \equiv \neg \Pi_L \Rightarrow L \text{ non regolare}$$

Teorema 2.8 (Pumping lemma per linguaggi regolari). Consideriamo un linguaggio regolare L , allora esiste una costante $k \in \mathbb{N}$ tale che per ogni stringa di lunghezza maggiore di k nel linguaggio, esiste una suddivisione della stringa in tre parti u, v, w tale che:

$$\begin{aligned}\exists k \in \mathbb{N} . \forall z \in L : |z| \geq k \\ \downarrow \\ \exists u, v, w \in \Sigma^* . z = uvw \wedge \begin{cases} |uv| \leq k \\ |v| > 0 \\ \forall i \in \mathbb{N} . uv^i w \in L \end{cases}\end{aligned}$$

cioè la parte v può essere "pompata" (ripetuta i volte) e la stringa risultante appartiene ancora a L .

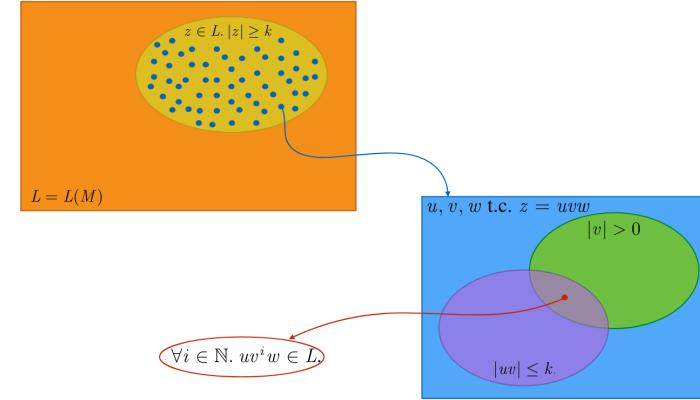


Figura 14: Rappresentazione grafica dell'insieme per il Pumping lemma

Dimostrazione:

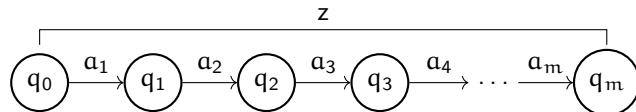
Consideriamo L regolare, allora esiste un automa DFA M tale che $L = L(M)$

$$M = \langle Q, \Sigma, \delta, q_0, F \rangle \quad |Q| = n \in \mathbb{N} \text{ stati finiti}$$

Vogliamo dimostrare che il lemma vale per $k = n$ (k è $|Q|$). Prendiamo una stringa nel linguaggio $z \in L$ tal che la sua lunghezza sia maggiore o uguale di k : $|z| \geq k$. Scriviamo z come insieme di caratteri:

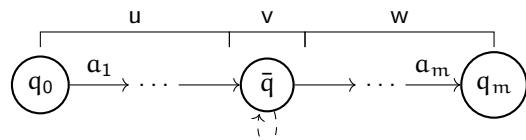
$$z = a_1 a_2 a_3 \dots a_m \quad m \geq k$$

Rappresentiamo l'elaborazione di z come una sequenza di stati:



Per leggere m simboli si attraversano $m + 1$ stati e quindi per **riconoscere** z si usano $m + 1$ stati, ma $m \geq n$, quindi si attraversano almeno $n + 1$ stati $\Rightarrow m + 1 \geq n + 1$. Questo implica che si attraversano più stati di quelli in Q , ovvero **almeno** uno stato è ripetuto nel riconoscimento di z .

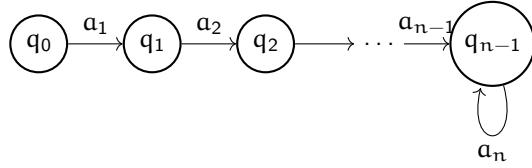
Supponiamo che $\bar{q} \in Q$ sia il primo stato (leggendo z) che viene **ripetuto**, in cui si torna per riconoscere z :



- u va da q_0 alla prima occorrenza di \bar{q} (il primo stato ripetuto)
- v va da \bar{q} a \bar{q} (percorso che porta a ritrovare lo stato ripetuto)
- w va da \bar{q} allo stato finale q_m

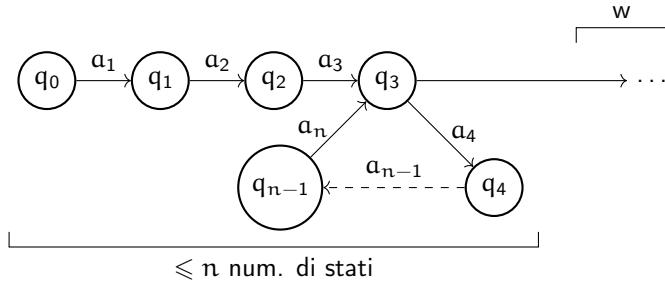
Questa suddivisione è accettabile per il teorema?

$$1. |uv| \leq k = n$$



Per arrivare allo stato finale q_n si attraversano $n - 1$ simboli

$$Q = \{q_0, q_1, \dots, q_{n-1}\} \rightarrow |Q| = n$$



2. $|v| > 0$ almeno un arco deve essere uscire dallo stato e ritornare nello stesso stato

3. uw arrivano in q_m , quindi:

$$\forall i . uv^i w \in L$$

e raggiunge lo stato finale q_m

2.6.5 Dimostrazione che un linguaggio non è regolare

Per dimostrare che un linguaggio non è regolare si può usare il pumping lemma:

$$L \text{ regolare} \Rightarrow \exists k . \forall z \in L : |z| \geq k$$

↓

$$\exists z = uvw . \begin{cases} |uv| \leq k \\ |v| > 0 \\ \forall i \in \mathbb{N} . uv^i w \in L \end{cases}$$

Per fare ciò si nega il pumping lemma:

$$A \rightsquigarrow B$$

$$\neg B \rightsquigarrow \neg A$$

$$\exists \rightsquigarrow \forall$$

Il pumping lemma si nega come segue:

- $\exists k \rightsquigarrow \forall K$: La dimostrazione **non** deve imporre vincoli su k
- $\forall z \rightsquigarrow \exists z \in L . |z| \geq k$: costruiamo noi la $z \in L$ di lunghezza $\geq k$
- $\exists uvw = z \rightsquigarrow \forall uvw = z : |v| > 0 |uv| \leq k$
- $\forall i \in \mathbb{N} \rightsquigarrow \exists i \in \mathbb{N} . uv^i w \notin L$: troviamo un i che "rompe" la stringa

Quindi il pumping lemma negato diventa:

$$\forall k \in \mathbb{N} . \exists z \in L : |z| \geq k$$

\Downarrow

$$\forall uvw = z . \begin{cases} |uv| \leq k \\ |v| > 0 \\ \exists i \in \mathbb{N} . uv^i w \notin L \end{cases}$$

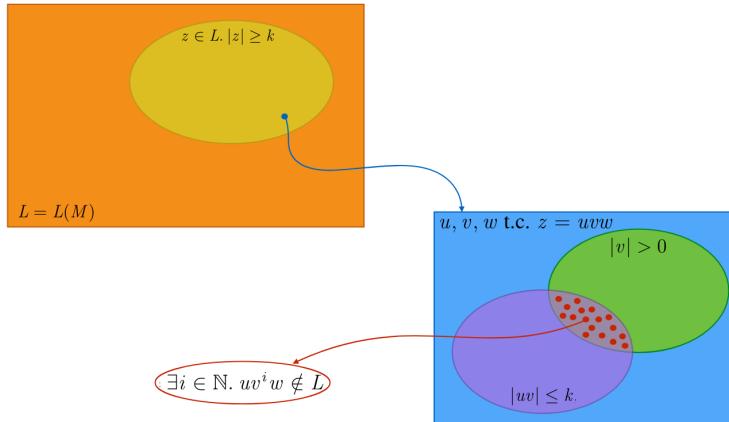


Figura 15: Rappresentazione grafica dell'insieme per il Pumping lemma negato

Esempio 2.10. Consideriamo il linguaggio:

$$L = \{0^n 1^n \mid n \in \mathbb{N}\}$$

Bisogna creare un automa che riconosca tutte le stringhe, ad esempio:

$$\epsilon, 01, 0011, 000111, 00001111, \dots$$

Scriviamo le condizioni di appartenenza a L :

$$0^a 1^b \in L \iff a = b$$

(gli 0 devono essere uguali agli 1).

Ogni volta che si legge un 1 bisogna ricordarsi quanti 0 sono stati letti prima, però si possono leggere infiniti 0 e quindi servirebbero infiniti stati. Vogliamo quindi dimostrare che L non è regolare usando il pumping lemma negato:

$$\forall k \in \mathbb{N} . \exists z \in L : |z| \geq k$$

\Downarrow

$$\forall uvw = z . \begin{cases} |uv| \leq k \\ |v| > 0 \\ \exists i \in \mathbb{N} . uv^i w \notin L \end{cases}$$

Fissiamo $k \in \mathbb{N}$ (k non deve avere nessun vincolo). Qualunque sia k prendiamo la stringa:

$$z = 0^k 1^k \in L \quad |z| \geq k$$

Le uniche suddivisioni che vanno bene sono quelle in cui uv stanno nella parte degli 0 (perchè per ipotesi $|uv| \leq k$. Si può concludere che la sottostringa uv è composta da soli 0:

$$uv \in 0^k$$

Consideriamo la stringa:

$$z_i = uv^i w = 0^{k+(i-1)|v|} 1^k$$

(ripetere v i volte equivale ad aggiungere $(i-1)|v|$ volte v a quella già esistente). Ad esempio: Supponendo di avere 10 zeri un esempio di suddivisione è il seguente

$$\underbrace{000000}_u \underbrace{000}_v \underbrace{0111111111}_{w}$$

- Con $i = 0$ (togliere v)

$$0^{k+(0-1)|v|} 1^k = 0^{k-|v|} 1^k$$

- Con $i = 1$ (lasciare v una volta)

$$0^{k+(1-1)|v|} 1^k = 0^k 1^k$$

- Con $i = 3$ (ripetere v tre volte)

$$0^{k+(3-1)|v|} 1^k = 0^{k+2|v|} 1^k$$

Troviamo un i tale che $z_i \notin L$:

$$z_i = 0^{k+(i-1)|v|} 1^k$$

Scegliamo ad esempio $i = 2$:

$$\begin{aligned} z_2 = 0^{k+|v|} 1^k \in L &\iff \underbrace{k + |v|}_{\alpha} = \underbrace{k}_{\beta} \\ &\iff |v| = 0 \text{ che è assurdo perch\`e } |v| > 0 \implies z_2 \notin L \end{aligned}$$

3 Linguaggi context free

I linguaggi context free sono più potenti dei linguaggi regolari, mantenendo comunque la decidibilità di molti problemi. Lo strumento che si utilizza per **definire** i linguaggi context free sono le **grammatiche**. Rispetto ai linguaggi regolari in cui l'automa era un riconoscitore, per i linguaggi context free la grammatica è un generatore. Un esempio di linguaggi context free sono i linguaggi di programmazione.

3.1 Grammatiche context free

Una grammatica context free è una quadrupla $E = \langle V, T, P, S \rangle$ dove:

- V è un insieme **finito** di simboli **non terminali** (variabili). Il loro ruolo è quello di rappresentare una categoria sintattica (ad esempio: espressione, istruzione, comando, ecc.)
- T è un insieme finito di simboli **terminali**. Sono i simboli effettivi che possono essere sostituiti a quelli non terminali. Ad esempio:

$$\underbrace{x}_{\text{terminale}} = \underbrace{\text{espressione}}_{\text{non terminale}}$$

- P è un insieme finito di **produzioni** (regole di derivazione). Indicano come sostituire i simboli non terminali con stringhe di simboli.
- $S \in V$ è il simbolo iniziale
- il **tipo di produzioni** (come possono essere fatte) determina il tipo di grammatica. In una **grammatica context free** le produzioni sono della forma:

$$A \rightarrow \alpha$$

dove $A \in V$ (non terminali) e $\alpha \in (V \cup T)^*$ (sequenza di simboli terminali e non terminali). Il vincolo principale è il fatto che a sinistra dell'implicazione ci sia esattamente **una sola variabile**. Questa implicazione significa che A può essere sostituito con α , indipendentemente dal contesto in cui A si trova, cioè dai simboli presenti prima e dopo A .

Esempio 3.1. Un esempio di grammatica sono le **espressioni booleane**. Consideriamo la grammatica: $G = \langle V, T, P, S \rangle$ dove:

$$V = \{E\}$$

$$T = \{0, 1, \text{and}, \text{or}, \text{not}, (,)\}$$

$$P = \begin{cases} E \rightarrow 0 \mid 1 \equiv E \rightarrow 0 \quad E \rightarrow 1 \\ E \rightarrow (E \text{ and } E) \\ E \rightarrow (E \text{ or } E) \\ E \rightarrow \text{not } E \end{cases} = E \rightarrow 0 \mid 1 \mid (E \text{ and } E) \mid (E \text{ or } E) \mid (\text{not } E)$$

$$S = E$$

(| è l'operatore or). Una grammatica può essere scritta anche solamente con l'insieme delle sue produzioni.

Il **linguaggio generato** dalla grammatica G è una combinazione di produzioni. Ad esempio:

$$A \rightarrow \beta \in P \quad \alpha, \gamma \in (V \cup T)^*$$

dove:

$$\begin{aligned} A &\in V \\ \beta &\in (V \cup T)^* \end{aligned}$$

Si deriva la stringa a destra applicando la produzione $A \rightarrow \beta$

$$\alpha A \gamma \Rightarrow \alpha \beta \gamma$$

Si è generata una nuova stringa sostituendo A con β .

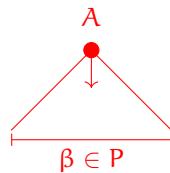


Figura 16: Rappresentazione grafica della produzione $A \rightarrow \beta$

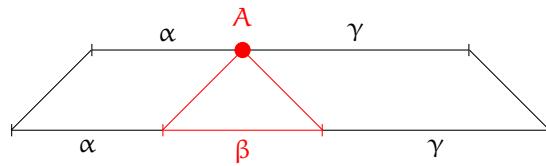


Figura 17: Rappresentazione grafica della derivazione $\alpha A \gamma \Rightarrow \alpha \beta \gamma$

La singola freccia (\rightarrow) indica una produzione, mentre la doppia freccia (\Rightarrow) indica una derivazione (applicazione di una produzione). Altri esempi di derivazioni:

- $E \rightarrow (E \text{ or } E) \equiv E \Rightarrow (E \text{ or } E)$
- $E \rightarrow (E \text{ and } E) \equiv \underbrace{(}_{\alpha} \underbrace{E}_{A} \underbrace{\text{ or } E}_{\gamma}) \Rightarrow \underbrace{(}_{\alpha} \underbrace{(E \text{ and } E)}_{\beta} \underbrace{\text{ or } E}_{\gamma})$

Definizione 3.1 (Derivazione). Partendo da una sequenza $\alpha \in V \cup T$ una derivazione consiste nell'applicazione di una produzione ad uno dei simboli non terminali presenti nella sequenza α . Considerando la stringa:

$$\alpha_1, \alpha_2, \alpha_3, \dots, \alpha_n \in (V \cup T)^*$$

e

$$\forall j \in [1, n-1] . \alpha_j \Rightarrow \alpha_{j+1}$$

allora si può scrivere l'applicazione di n produzioni come:

$$\alpha_1 \Rightarrow_n \alpha_n$$

Ad esempio:

$$\alpha_i \Rightarrow_0 \alpha_j \quad \text{Nessuna produzione applicata}$$

Si può scrivere la chiusura transitiva di \Rightarrow_n come:

$$\alpha \Rightarrow^* \beta \text{ se } \exists i . \alpha \Rightarrow_i \beta$$

Cioè esiste un numero di passi i tale che applicando i produzioni si passa da α a β . (Da α si deriva β)

Definizione 3.2 (Linguaggio generato). Il linguaggio L è generato dalla grammatica $G : L(G)$ con $G = \langle V, T, P, S \rangle$ è l'insieme delle sequenze di terminali generate da S , ovvero per le quali esiste una derivazione a partire da S :

$$L(G) = \left\{ \sigma \in T^* \mid S \xrightarrow{G}^* \sigma \right\}$$

Il linguaggio $L \subseteq T^*$ è **context free** se esiste una grammatica G context free che lo genera, ovvero tale che $L = L(G)$. Quindi per dimostrare che un linguaggio è context free basta costruire una grammatica che lo genera.

3.1.1 Dimostrazione canonica che un linguaggio è context free

Definizione 3.3 (Dimostrazione che un linguaggio è context free). Possiamo dimostrare che $L = L(G)$, quindi L è context free. Dimostriamo:

$$1. \quad L \subseteq L(G)$$

Se σ appartiene al linguaggio, allora da S si può derivare σ :

$$L \subseteq L(G) \equiv \sigma \in L \implies \sigma \in L(G) \equiv S \Rightarrow^* \sigma$$

Dimostriamo per induzione sulla lunghezza di σ .

- **Caso base:** $|\sigma| =$ lunghezza minima in L
- **Passo induttivo:** Si suppone che per ogni σ di lunghezza n valga l'ipotesi induttiva, e si dimostra che vale anche per σ di lunghezza maggiore di n .

$$\forall \sigma . |\sigma| = n \text{ vale l'ipotesi induttiva 1.}$$

$$2. \quad L(G) \subseteq L$$

Se σ appartiene al linguaggio generato dalla grammatica, allora σ appartiene al linguaggio:

$$L(G) \subseteq L \equiv \sigma \in L(G) \implies \sigma \in L \equiv S \Rightarrow^* \sigma$$

Dimostriamo per induzione sulla lunghezza della derivazione:

$$S \Rightarrow^* \sigma \equiv \exists i . S \Rightarrow_i \sigma$$

quindi l'induzione è su i .

- **Caso base:** La derivazione più corta possibile di simboli terminali
- **Passo induttivo:** Si suppone che per ogni $i \leq n$ se la S deriva in i passi in una stringa di soli terminali σ allora $\sigma \in L$:

$$\forall i \leq n . S \Rightarrow_i \sigma \text{ allora } \sigma \in L$$

Dimostriamo che $S \Rightarrow_{n+1} \sigma$ allora $\sigma \in L$.

Esempio 3.2. Consideriamo il seguente linguaggio:

$$L = \{0^n 1^n \mid n \in \mathbb{N}\}$$

Vogliamo dimostrare che L è context free. Per farlo bisogna costruire una grammatica G che genera L :

- S: Bisogna chiedersi se la stringa vuota appartiene a L.

1. $\epsilon \in L$? Si, per $n = 0$ $0^0 1^0 = \epsilon$. ϵ va sempre generata dalla grammatica.

Quindi S diventa:

$$S \rightarrow \epsilon | 0S1$$

La grammatica di questo linguaggio quindi è:

$$G : S \rightarrow \epsilon | 0S1 \text{ (grammatica context free)}$$

Dimostriamo: $L = L(G)$

1. $L \subseteq L(G)$ per induzione sulla lunghezza di $\sigma \in L$

- **Tesi:** $\sigma \in L$ allora $S \Rightarrow^* \sigma$
- **Base:** $n = 0$ $|\sigma| = 0$ è tale che $\sigma = \epsilon$ e $\sigma \rightarrow \epsilon$ ($S \Rightarrow \epsilon$)
- **Passo induttivo:** L'ipotesi induttiva è la tesi, ma limitata a n:

$$\forall i \leq n . |\sigma| = i \sigma \in L \text{ allora } S \Rightarrow^* \sigma$$

Dimostriamo che $|\sigma| = n$ e $\sigma \in L$ allora $S \Rightarrow^* \sigma$. Prendiamo $\sigma \in L$, ovvero:

$$\exists j . \sigma = 0^j 1^j \quad 2j = n$$

Bisogna trovare la derivazione. Possiamo osservare che:

$$\sigma = 0 \underbrace{0^{j-1} 1^{j-1}}_{2j-2 < n} 1$$

(abbiamo separato il primo 0 e l'ultimo 1 dalla sequenza centrale). σ può essere scritta come:

$$\sigma = 0 \underbrace{0^{j-1} 1^{j-1}}_{2j-2 < n} 1 = 0\sigma'1 \quad \text{con } \sigma' = 0^{j-1} 1^{j-1} \in L$$

Siccome $|\sigma'| < n$ e $\sigma' \in L$ allora vale l'ipotesi induttiva e quindi:

$$\implies \exists S \Rightarrow^* \sigma' \equiv 0^{j-1} 1^{j-1}$$

usiamo questa derivazione per costruire la derivazione di σ . Questa derivazione deve essere fatta in modo da avere come ultimo passo $S \Rightarrow \epsilon$ per ottenere σ' :

$$S \Rightarrow^* 0^{j-1} 1^{j-1} \equiv S \Rightarrow 0^{j-1} S 1^{j-1} \Rightarrow 0^{j-1} \epsilon 1^{j-1} = 0^{j-1} 1^{j-1}$$

Per arrivare a $\sigma = 0^j 1^j$ bisogna sostituire la S con $0S1$:

$$S \Rightarrow^* 0^j S 1^j \Rightarrow 0^{j-1} 0 S 1 1^{j-1} \Rightarrow 0^{j-1} 0 \varepsilon 1 1^{j-1} = 0^j 1^j = \sigma \quad \square$$

Questo conclude la dimostrazione del primo punto, cioè trovare una derivazione per σ . Quindi abbiamo dimostrato che $L \subseteq L(G)$.

2. $L(G) \subseteq L$ Dimostriamo per induzione sulla lunghezza delle derivazioni

- **Tesi:** $S \Rightarrow^* \sigma$ allora $\sigma \in L$
- **Base:** $S \Rightarrow \varepsilon$ ($S \rightarrow \varepsilon$) e $\varepsilon \in L$
- **Passo induttivo:** L'ipotesi induttiva è la tesi limitata a i:

$$\forall i \leq n . S \Rightarrow_i \sigma \text{ allora } \sigma \in L$$

Prendiamo una stringa generata in n passi:

$$S \Rightarrow_n \sigma \equiv S \Rightarrow_{n-1} \sigma_1 S \sigma_2 \Rightarrow \sigma_1 \sigma_2 = \sigma$$

Per come è fatta la grammatica σ_1 contiene solo 0 e σ_2 contiene solo 1. Ovvvero:

$$\sigma_1 = 0^j \quad \sigma_2 = 1^k$$

Bisogna riuscire ad ottenere una derivazione più corta di n simboli terminali in modo tale da poter applicare l'ipotesi induttiva. Un solo passo indietro non basta perché non si ha una derivazione di tutti simboli terminali, quindi si può andare ancora più indietro:

$$S \Rightarrow_{n-1} \sigma_1 S \sigma_2 \equiv S \Rightarrow_{n-2} \sigma'_1 S \sigma'_2 \Rightarrow \underbrace{\sigma'_1}_{{\sigma_1}} \underbrace{0}_{{\color{red} S}} \underbrace{1 \sigma'_2}_{{\sigma_2}}$$

Scorriamo la derivazione di $n - 2$ passi in:

$$S \Rightarrow_{n-2} \sigma'_1 S \sigma'_2 \Rightarrow \sigma'_1 \sigma'_2$$

abbiamo una derivazione lunga $n - 1$ di simboli terminali dove:

$$\begin{aligned} \sigma_1 &= \sigma'_1 0 \implies \sigma'_1 = 0^{h-1} \\ \sigma_2 &= 1 \sigma'_2 \implies \sigma'_2 = 1^{k-1} \end{aligned}$$

Per ipotesi induttiva:

$$\xrightarrow{\text{I.I.}} \sigma'_1 \sigma'_2 \in L$$

Quindi:

$$\sigma'_1 \sigma'_2 = 0^{h-1} 1^{k-1} \in L \implies h-1 = k-1 \implies h = k$$

ma allora $\sigma = \sigma_1 \sigma_2 = 0^h 1^k$ è tale che $h = k$ e quindi $\sigma \in L$

Ricapitolando:

- Si parte da $S \Rightarrow_n 0^h 1^k$, ma non si può applicare l'ipotesi induttiva, quindi si va un passo indietro
- $S \Rightarrow_{n-1} 0^h S 1^k$, ma ancora non si può applicare l'ipotesi induttiva
- Si va ancora un passo indietro:

$$S \Rightarrow_{n-2} 0^{h-1} S 1^{k-1} \Rightarrow 0^{h-1} 0 S 1^{k-1} \Rightarrow 0^h 1^k$$

questo si può usare per generare $0^{h-1} 1^{k-1}$:

$$S \Rightarrow_{n-2} 0^{h-1} S 1^{k-1} \Rightarrow 0^{h-1} 1^{k-1}$$

che è lunga $n-1$ e quindi si può applicare l'ipotesi induttiva.

- Quindi $0^{h-1} 1^{k-1} \in L \implies h-1 = k-1 \implies h = k$ e quindi $0^h 1^k \in L \quad \square$

Siamo andati indietro tanto quanto serviva per generare una stringa di soli simboli terminali.

3.1.2 Dimostrazione alternativa

Esempio 3.3. Considerando l'esempio precedente, una dimostrazione alternativa che un linguaggio è context free consiste nel partire dall'inizio e fare passi avanti piuttosto che partire dalla fine e fare passi indietro. Questo non funziona sempre, ma in questo caso sì:

$$L(G) \subseteq L$$

$$\forall i < n . S \Rightarrow_i \sigma \text{ allora } \sigma \in L$$

- **Passo induttivo:**

$$S \Rightarrow_n \sigma \equiv S \Rightarrow 0S1 \Rightarrow_{n-1} \sigma \equiv 0\sigma'1$$

dove $S \Rightarrow_{n-1} \sigma'$. Ma allora si può applicare l'ipotesi induttiva e quindi $\sigma' \in L$ e questo significa che:

$$\exists j . \sigma' = 0^j 1^j$$

questo implica che:

$$\sigma = 0\sigma'1 = 00^j 1^j 1 = 0^{j+1} 1^{j+1} \in L$$

3.2 Alberi di derivazione

Data una grammatica context free $G = \langle V, T, P, S \rangle$, un albero di derivazione (parse tree) per G ha le seguenti caratteristiche:

1. Ogni nodo ha un'etichetta che è un simbolo di $V \cup T \cup \{\epsilon\}$
2. L'etichetta della radice è un simbolo in V

3. Ogni nodo interno interno (non foglia) ha etichetta in V
4. Se un nodo n è etichettato con $A \in V$ e i nodi figli n_1, n_2, \dots, n_k sono etichettati con $X_1, X_2, \dots, X_k \in V \cup T \cup \{\varepsilon\}$ allora esiste una produzione che genera un arco per ogni simbolo figlio:

$$A \rightarrow X_1 X_2 \dots X_k \in P$$

5. Se un nodo ha etichetta ε allora è una foglia ed è l'unico figlio del padre

Esempio 3.4. Consideriamo la grammatica

$$E \rightarrow 0 \mid 1 \mid (E \text{ and } E) \mid (E \text{ or } E) \mid (\text{not } E)$$

Consideriamo la derivazione:

$$E \Rightarrow^* ((0 \text{ or } 1) \text{ and } (\text{not } 0))$$

L'albero di derivazione è il seguente:

Teorema 3.1. Sia $G = \langle V, T, P, S \rangle$ una grammatica context free, allora:

$$S \Rightarrow^* \alpha \in T^*$$

se e solo se esiste un albero di derivazione con radice etichettata S e foglie etichettate (leggendo da sinistra verso destra) con i simboli di α .

3.3 Ambiguità delle grammatiche

Esempio 3.5. Consideriamo una grammatica per le espressioni aritmetiche senza parentesi:

$$E \rightarrow E + E \mid E * E \mid 0 \mid 1 \mid 2$$

Consideriamo la stringa:

$$2 * 0 + 1$$

Si hanno due casi:

- Si esegue prima la moltiplicazione:

$$2 * 0 + 1 = (2 * 0) + 1 = 1$$

- Si esegue prima l'addizione:

$$2 * 0 + 1 = 2 * (0 + 1) = 2$$

Notiamo che si ottengono due risultati diversi, quindi la grammatica è ambigua perché la stessa stringa può essere generata da due alberi di derivazione distinti:

- Albero di derivazione con precedenza alla moltiplicazione
- Albero di derivazione con precedenza all'addizione

Definizione 3.4 (Ambiguità). Una grammatica è **ambigua** se esiste almeno una parola α con più di un albero di derivazione con radice S .

Il linguaggio generato da una grammatica ambigua è detto **inherentemente ambiguo**.

3.4 Forma normale

Le forme normali sono grammatiche le cui produzioni rispettano vincoli di "forma" specifici. Ci sono due forme normali importanti per le grammatiche context free:

- **Forma normale di Chomsky:** Tutte le produzioni hanno la forma:

$$A \rightarrow a \quad \vee \quad A \rightarrow BC$$

dove:

$$A, B, C \in V \quad a \in T$$

Questa forma genera sempre un albero binario (ogni nodo ha al massimo due figli).

- **Forma normale di Greibach:** Ogni linguaggio context free è generato da una grammatica le cui produzioni sono della forma

$$A \rightarrow a\alpha$$

dove:

$$A \in V \quad a \in T \quad \alpha \in V^*$$

Rappresenta l'automa a pila in cui α è il contenuto della pila e a è il simbolo da aggiungere.

3.5 Conversione in forma normale di Chomsky

Ogni grammatica context free può essere riscritta in modo tale che:

1. **Eliminazione dei simboli inutili:** Ogni simbolo non terminale genera simboli terminali e ogni simbolo terminale è generato da almeno un simbolo non terminale
2. **Eliminazione delle produzioni unitarie:** Nessuna produzione è della forma:

$$A \rightarrow B \quad A, B \in V$$

perchè A è generabile da B e B è generabile da A .

3. **Eliminazione della ε -produzione:** Se $\varepsilon \notin L$ allora **non** ci deve essere la produzione:

$$A \rightarrow \varepsilon \quad A \in V$$

Definizione 3.5. Quando la grammatica è senza simboli inutili, senza produzioni unitarie e senza ϵ -produzioni, si può trasformare in forma normale di Chomsky.

3.5.1 Eliminazione dei simboli inutili

$x \in V \cup T$ è utile se esiste una derivazione

$$S \Rightarrow^* \underbrace{\alpha x \beta}_{1.a} \Rightarrow^* w \in T^*$$

Teorema 3.2. Per ogni grammatica context free $\forall G = \langle V, T, P, S \rangle$ esiste una grammatica equivalente $G' = \langle V', T, P', S \rangle$ senza simboli inutili:

$$\forall G . \exists G' . L(G) = L(G') \text{ e } G' \text{ non ha simboli inutili}$$

G' si ottiene applicando in sequenza le due trasformazioni viste (1, 2) (nell'ordine visto)

1. Eliminiamo i simboli che non generano terminali:

Esempio 3.6. Consideriamo le produzioni $S \rightarrow \alpha AB$ e $A \rightarrow A$. Il simbolo A è inutile perché non genera mai simboli terminali.

Metodo algoritmico per eliminare i simboli che non generano terminali:

- Dopo la trasformazione

$$\forall A \in V . \exists w \in T^* . A \Rightarrow^* w$$

Allora $L(G) = \emptyset$ tutti i simboli sono inutili

- Prendiamo per ipotesi che $L(G) \neq 0$, definiamo:

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

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

Esempio 3.7. Consideriamo le seguenti produzioni:

$$\begin{aligned} A &\rightarrow A \\ S &\rightarrow aS \mid B \\ B &\rightarrow b \end{aligned}$$

Si parte con $W = \emptyset$:

- $\Gamma^0(\emptyset) = \{A \in V \mid \exists \alpha \in T^* . A \rightarrow \alpha \in P\} = \emptyset$
- $\Gamma^1(\emptyset) = \Gamma(\emptyset) = \{B\}$ perché $B \rightarrow b \in P$

- $\Gamma^2(\emptyset) = \Gamma(\{B\}) = \{A \in V \mid \exists \alpha \in (T \cup \{B\})^* . A \rightarrow \alpha\} = \{B, S\}$
- $\Gamma^3(\emptyset) = \Gamma(\{B, S\}) = \{A \in V \mid \exists \alpha \in (T \cup \{B, S\})^* . A \rightarrow \alpha\} = \{B, S\}$
Si è raggiunto il **punto fisso**, cioè non si ottengono più simboli nuovi.

Alla fine si ottiene l'insieme di tutti i simboli utili, cioè quelli che generano simboli terminali. Tutti gli altri (e le corrispondenti produzioni) vengono eliminati.

Quindi $S \rightarrow aS \mid B \quad B \rightarrow b$ è equivalente ma senza simboli inutili che non generano nulla

2. Eliminiamo i simboli non raggiungibili da S :

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

Quindi ad esempio:

$$\Gamma(\emptyset) = \{S\}$$

Cioè si aggiunge ciò che possiamo raggiungere da S in un certo numero di passi.

Esempio 3.8. Consideriamo le seguenti produzioni:

$$\begin{aligned} S &\rightarrow a \mid bC \\ C &\rightarrow d \mid dE \\ E &\rightarrow \epsilon \\ F &\rightarrow f \end{aligned}$$

Si parte con $W = \emptyset$:

- $\Gamma(\emptyset) = \{S\}$
- $\Gamma(\{S\}) = \{X \in V \cup T \mid S \rightarrow \alpha X \beta \in P\} \cup \{S\} = \{S, a, b, C\}$
- $\Gamma(\{S, a, b, C\}) = \{X \in V \cup T \mid C \rightarrow \alpha X \beta \in P \vee S \rightarrow \alpha X \beta \in P\} = \{S, a, b, C, d, E\}$
- $\Gamma(\{S, a, b, C, d, E\}) = \{S, a, b, C, d, E, \epsilon\}$

Quindi F è un simbolo non raggiungibile da S e quindi è inutile.

3.5.2 Eliminazione delle produzioni unitarie

L'idea è quella di sostituire le produzioni unitarie con le produzioni dello stesso simbolo non terminale a cui puntano. Ad esempio se abbiamo $S \rightarrow A \rightarrow 0A0$ si può compattare S come: $S \rightarrow 0A0$

Esempio 3.9. Consideriamo la seguente grammatica:

$$\begin{aligned} S &\rightarrow \varepsilon \mid A \mid B \\ A &\rightarrow 0 \mid 0A0 \mid B \mid 00 \\ B &\rightarrow 1 \mid 1B1 \mid A \mid 11 \end{aligned}$$

Sostituiamo in tutte le produzioni quello che è producibile dal non terminale a destra:

$$\begin{aligned} S &\rightarrow \varepsilon \mid 0 \underbrace{| 0A0 | 00}_{A} \mid 1 \underbrace{| 1B1 | 11}_{B} \\ A &\rightarrow 0 \mid 0A0 \mid 00 \mid 1 \underbrace{| 1B1 | 11}_{B} \\ B &\rightarrow 1 \mid 1B1 \mid 11 \mid 0 \underbrace{| 0A0 | 00}_{A} \end{aligned}$$

3.5.3 Eliminazione delle ε -produzioni

L'idea è quella di sostituire la transizione con tutto quello che può generare ε ad esempio:

Esempio 3.10. Consideriamo le produzioni:

$$\begin{aligned} S &\rightarrow \varepsilon \mid A \mid B \\ A &\rightarrow 0 \mid 0A0 \mid B \\ B &\rightarrow 1 \mid 1B1 \mid A \mid \varepsilon \end{aligned}$$

Siccome sia S che B vanno in ε si possono sostituire con:

$$B \rightarrow S$$

Anche A con un certo numero di derivazioni arriva in ε , quindi tutti i simboli non terminali che in un certo numero di passi generano ε sono:

$$\{S, A, B\}$$

Le nuove produzioni diventano:

$$\begin{aligned} S &\rightarrow \varepsilon \mid A \mid B \\ A &\rightarrow 0 \mid 0A0 \mid 00 \mid B \\ B &\rightarrow 1 \mid 1B1 \mid 11 \mid A \end{aligned}$$

Eliminando $B \rightarrow \varepsilon$ le stringhe 00 e 11 non sarebbero più generabili e quindi si aggiungono

In questo modo si sono eliminate tutte le ε -produzioni, tranne $S \rightarrow \varepsilon$ se $\varepsilon \in L$.

3.6 Trasformazione di una grammatica in forma normale di Chomsky

Quando una grammatica non ha simboli inutili, non ha produzioni unitarie e non ha ϵ -produzioni, si può trasformare in forma normale di Chomsky, cioè tutte le produzioni hanno la forma:

$$A \rightarrow a \in T \quad \vee \quad A \rightarrow BC \quad B, C \in V$$

Esempio 3.11. Consideriamo la seguente grammatica:

$$\begin{aligned} S &\rightarrow \epsilon \\ S &\rightarrow 0 \mid 1 \\ S &\rightarrow 00 \mid 0S0 \mid 11 \mid 1S1 \end{aligned}$$

Trasformiamo le sequenze più lunghe di 1 simbolo terminale in simboli non terminali:

$$\begin{aligned} S &\rightarrow \epsilon \\ S &\rightarrow 0 \mid 1 \\ S &\rightarrow V_1V_1 \mid V_1SV_1 \mid V_2V_2 \mid V_2SV_2 \\ V_1 &\rightarrow 0 \\ V_2 &\rightarrow 1 \end{aligned}$$

Trasformiamo le sequenze più lunghe di 2 simboli non terminali in un singolo simbolo non terminale:

$$\begin{aligned} S &\rightarrow \epsilon \\ S &\rightarrow 0 \mid 1 \\ S &\rightarrow V_1V_1 \mid V_3V_1 \mid V_2V_2 \mid V_4V_2 \\ V_1 &\rightarrow 0 \\ V_2 &\rightarrow 1 \\ V_3 &\rightarrow V_1S \\ V_4 &\rightarrow V_2S \end{aligned}$$

Ora tutte le produzioni sono in forma normale di Chomsky.

Questa forma è semplicemente una forma canonica facilmente rappresentabile, non una forma più semplice.

3.7 Transformazione in forma normale di Greibach

Per trasformare una grammatica in forma normale di Greibach si fa un **unfolding**. Consideriamo una grammatica $G = \langle V, T, P, S \rangle$ definita da:

$$\begin{aligned} A &\rightarrow \alpha B \gamma \in P \\ B &\rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n \in P \end{aligned}$$

Allora $G' = \langle V, T, P', S \rangle$ dove eliminiamo la produzione $A \rightarrow \alpha B \gamma$ che non è nella forma corretta e aggiungiamo con la sostituzione:

$$A \rightarrow \alpha \beta_1 \gamma \mid \alpha \beta_2 \gamma \mid \dots \mid \alpha \beta_n \gamma \in P'$$

Questa trasformazione non cambia il linguaggio generato:

$$L(G) = L(G')$$

Esempio 3.12.

$$S \rightarrow aSb \mid ab$$

$$B \rightarrow b$$

$$S \rightarrow aSB \mid aB$$

Teorema 3.3 (Eliminazione ricorsiva sinistra). Consideriamo la grammatica $G = \langle V, T, P, S \rangle$ con:

$$\begin{aligned} A \rightarrow A\alpha_1 \mid \dots \mid A\alpha_n &\quad \rightsquigarrow A \rightarrow Aa \\ A \rightarrow \beta_1 \mid \dots \mid \beta_n &\quad \rightsquigarrow A \rightarrow b \end{aligned}$$

Se costruiamo $G' = \langle V \cup \{B\}, T, P', S \rangle$ dove in P' sostituiamo le produzioni per A con:

$$\begin{aligned} A \rightarrow \beta_i \mid \beta_i B \\ B \rightarrow \alpha_i \mid \alpha_i B \end{aligned}$$

allora $L(G) = L(G')$.

3.8 Pumping lemma per i linguaggi context free

Il pumping lemma è una **condizione necessaria** per essere un linguaggio context free:

$$L \text{ è CF} \implies \text{vale } \Pi \text{ condizione del pumping lemma}$$

Teorema 3.4 (Pumping lemma per i linguaggi context free). Sia L un linguaggio context free. Allora esiste una costante k tale che per ogni stringa $z \in L$ con $|z| \geq k$ esiste una suddivisione di z in cinque sottostringhe:

$$\exists k \in \mathbb{N} . \forall z \in L . |z| \geq k \implies \exists u, v, w, x, y = z \vee \begin{cases} |vwx| \leq k \\ |vx| \geq 0 \\ \forall i \in \mathbb{N} . uv^i wx^i y \in L \end{cases}$$

3.8.1 Dimostrazione (grafica)

Consideriamo un linguaggio context free $L \subseteq T^*$, allora esiste una grammatica tale che $L = L(G)$:

$$L \subseteq T^* \implies \exists G = \langle V, T, P, S \rangle \text{ CF} . L = L(G)$$

Supponiamo, senza perdere generalità, che G sia in forma normale di Chomsky. Definiamo n come la dimensione di V , cioè il numero di simboli non terminali in G :

$$n = |V|$$

Poichè la grammatica è in forma normale di Chomsky questo implica che tutti gli alberi di derivazione in G sono binari, cioè ogni nodo ha due figli non terminali oppure una foglia terminale. L'altezza dell'albero è h e di conseguenza il numero di elementi ad una certa altezza è 2^{h-1} .

Definiamo $k = 2^n$ e prendiamo una stringa $z \in L$ di lunghezza maggiore o uguale a k :

$$z \in L . |z| \geq k$$

Se le foglie sono maggiori o uguali a 2^n allora l'altezza dell'albero è $\geq n + 1$. Il numero di **archi** che collegano simboli non terminali sono $\geq n$. Di conseguenza il numero di nodi interni (non terminali) nel cammino sono $n + 1$. Siccome in questa grammatica il numero di nodi non terminali è n , allora almeno una etichetta non terminale è ripetuta nel cammino due volte. (Ci sono almeno due nodi interni che hanno la stessa etichetta) Supponiamo di prendere il nodo A , più vicino alle foglie, che si ripete (implica che dalla prima occorrenza di A fino alle foglie non ci sono ripetizioni). Consideriamo il sottoalbero radicato in questo nodo e quello radicato nel secondo nodo con la stessa etichetta. Osserviamo che la stringa è stata divisa in cinque parti u, v, w, x, y : Valgono i seguenti vincoli:

- $|vx| \geq 0$: almeno un simbolo deve essere generato in v o x
- $|vwx| \leq k$: se dalla prima occorrenza di A fino alle foglie non ci sono ripetizioni, allora attraversiamo massimo $n = |V|$ nodi e quindi non si generano più di $2^n = k$ simboli terminali

3.8.2 Dimostrare che un linguaggio non è context free

Il pumping lemma può essere usato per dimostrare che un linguaggio è context free:

$$L \text{ è CF} \implies \text{vale } \Pi \text{ condizione del pumping lemma}$$

Si può negare il pumping lemma per ottenere una condizione **sufficiente** a dimostrare che un linguaggio non è context free:

$$\neg \Pi \text{ condizione del pumping lemma} \implies L \text{ non è CF}$$

dove Π è:

$$\Pi : \exists k \in \mathbb{N} . \forall z \in L . |z| \geq k . \exists u, v, w, x, y = z \vee \begin{cases} |vwx| \leq k \\ |vx| \geq 0 \end{cases}$$

$$\cdot \forall i \in \mathbb{N} . uv^iwx^i y \in L$$

La negazione di Π è:

$$\neg\Pi : \forall k \in \mathbb{N} . \exists z \in L . |z| \geq k . \forall u, v, w, x, y = z \vee \begin{cases} |vwx| \leq k \\ |vx| \geq 0 \end{cases}$$

$$\cdot \exists i \in \mathbb{N} . uv^iwx^i y \notin L$$

Esempio 3.13. Consideriamo il seguente linguaggio:

$$L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$$

Dimostriamo che L non è context free usando la negazione del pumping lemma.

Prendiamo $n = k$ siccome n non ha alcun vincolo e scegliamo la stringa:

$$z = a^k b^k c^k \in L$$

Tutte le possibili suddivisioni della stringa sono le seguenti: Le suddivisioni a cavallo di almeno due gruppi sono tali che $\forall i > 1$ cambiamo la struttura della stringa e quindi esce da L di conseguenza non le consideriamo. Bisognerebbe dimostrare tutte le suddivisioni, ma siccome le dimostrazioni per alcune sarebbero uguali non si ripetono.

Le condizioni di appartenenza a L sono:

$$a^i b^j c^l \in L \iff i = j = l$$

1. Se la suddivisione è $z = a^k b^k c^k \quad v, x \in a^k$:

$$z_i = a^{k+(i-1)|vx|} b^k c^k$$

• Consideriamo $i = 2$:

$$z_2 = a^{k+|vx|} b^k c^k$$

verifichiamo se appartiene a L :

$$z_2 \in L \iff \begin{cases} k + |vx| = k \iff |vx| = 0 \text{ (falso)} \\ k = k \text{ (ovvio)} \end{cases}$$

Poichè per costruzione $|vx| \geq 0$ allora $z_2 \notin L$

2. Se la suddivisione è $v \in a^k \quad x \in b^k$:

$$z_i = a^{k+(i-1)|v|} b^{k+(i-1)|x|} c^k$$

• Consideriamo $i = 2$:

$$z_2 = a^{k+|v|} b^{k+|x|} c^k$$

verifichiamo se appartiene a L :

$$\begin{aligned} z_2 \in L &\iff k + |v| = k + |x| = k \\ &\iff |v| = |x| = 0 \implies |vx| = 0 \text{ (falso)} \end{aligned}$$

poiche $|vx| > 0$ allora $z_2 \notin L$

Abbiamo dimostrato che per ogni suddivisione di z esiste un i tale che $z_i \notin L$, quindi L non è context free.

3.9 Proprietà di chiusura dei linguaggi context free

Teorema 3.5. I linguaggi context free sono chiusi per **unione, concatenazione e stella di Kleene**.

Consideriamo due grammatiche context free $G_1 = \langle V_1, T_1, P_1, S_1 \rangle$ e $G_2 = \langle V_2, T_2, P_2, S_2 \rangle$.

- Esiste una grammatica che genera l'unione delle due grammatiche:

$$L(G_1) \cup L(G_2) \text{ è CF}$$

Bisogna aggiungere un simbolo iniziale che andrà nei simboli iniziali delle due grammatiche:

$$G' = \langle V_1 \cup V_2 \cup \{S\}, T_1 \cup T_2, P_1 \cup P_2 \cup \{S \rightarrow S_1 \mid S_2\}, S \rangle$$

- Esiste una grammatica che genera la concatenazione delle due grammatiche:

$$L(G_1) \cdot L(G_2) \text{ è CF}$$

Bisogna aggiungere un simbolo iniziale che andrà nei simboli iniziali delle due grammatiche in sequenza:

$$G' = \langle V_1 \cup V_2 \cup \{S\}, T_1 \cup T_2, P_1 \cup P_2 \cup \{S \rightarrow S_1 S_2\}, S \rangle$$

- Esiste una grammatica che genera la stella di Kleene della grammatica:

$$L(G_1)^* \text{ è CF}$$

Bisogna aggiungere un simbolo iniziale che andrà nel simbolo iniziale della grammatica oppure in ϵ :

$$G' = \langle V_1 \cup \{S\}, T_1, P_1 \cup \{S \rightarrow S_1 S \mid \epsilon\}, S \rangle$$

Teorema 3.6. I linguaggi context free **non** sono chiusi per intersezione. Con-

sideriamo due linguaggi context free L_1 e L_2 , allora:

$L_1 \cap L_2$ potrebbe non essere CF

Dimostrazione: Consideriamo il seguente linguaggio:

$$L_1 = \{a^i b^i c^j \mid i, j \in \mathbb{N}\}$$

La grammatica che genera L_1 è:

$$G_1 = \begin{cases} S \rightarrow RC \\ R \rightarrow aRb \mid \epsilon & L_1 = L(G_1) \\ C \rightarrow cC \mid \epsilon \end{cases}$$

Consideriamo anche il seguente linguaggio:

$$L_2 = \{a^j b^i c^i \mid i, j \in \mathbb{N}\}$$

La grammatica che genera L_2 è:

$$G_2 = \begin{cases} S \rightarrow AR \\ A \rightarrow aA \mid \epsilon & L_2 = L(G_2) \\ C \rightarrow bCc \mid \epsilon \end{cases}$$

L'intersezione dei due linguaggi è:

$$\begin{aligned} L &= L_1 \cap L_2 \\ &= \{a^i b^j c^k \mid i = j, j = k\} \\ &= \{a^i b^i c^k \mid i = j = k\} \\ &= \{a^n b^n c^n \mid n \in \mathbb{N}\} \text{ non è CF} \end{aligned}$$

L'intersezione non è context free (dimostrato tramite pumping lemma) e quindi abbiamo dimostrato che i linguaggi context free non sono chiusi per intersezione.

Definizione utile 3.1. Le proprietà di **unione** e **intersezione** di linguaggi funzionano solo l'unione e l'intersezione **finita**.

Se L_i sono infiniti linguaggi regolari o context free, allora:

$$\bigcup_i L_i \text{ potrebbe non essere regolare o CF}$$

Teorema 3.7. I linguaggi context free non sono chiusi per **complemento**.

Dimostrazione: Dato $L \in CF$, se anche il complemento $\bar{L} \in CF$ fosse context

free, allora osserviamo che:

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}} \text{ (de Morgan)}$$

Poiché i linguaggi context free sono chiusi per unione e complemento, allora $\overline{L_1}$ e $\overline{L_2}$ sarebbero context free e di conseguenza anche la loro intersezione sarebbe context free:

$$\overline{L_1}, \overline{L_2} \in CF \implies \overline{L_1} \cup \overline{L_2} \in CF$$

Questo implica che anche l'intersezione sarebbe context free:

$$\implies \overline{\overline{L_1} \cup \overline{L_2}} \in CF \implies L_1 \cap L_2 \in CF$$

Abbiamo dimostrato però che l'intersezione di due linguaggi context free potrebbe non essere context free, quindi la nostra ipotesi che anche

3.10 Problemi decidibili per i linguaggi context free

Per i linguaggi context free i seguenti problemi sono decidibili, cioè esiste un algoritmo che risolve il problema in un numero finito di passi: Sia L un linguaggio context free:

1. $L = \emptyset$
2. L è finito
3. L è infinito
4. Determinare se una stringa appartiene al linguaggio $x \in L$, cioè se è generata dalla grammatica del linguaggio.

3.11 Automa per riconoscere i linguaggi context free (Automi a pila)

L'automa a pila è un automa con memoria ausiliaria (pila) che permette di riconoscere i linguaggi context free.

Esempio 3.14. Per il seguente linguaggio:

$$\{a^n b^n \mid n \in \mathbb{N}\}$$

Ci sarebbe bisogno di n stati per contare le occorrenze di a e b . Invece con un automa a pila si può usare la pila per contare le occorrenze di a e b

Definizione 3.6. L'APND (Automa a pila non deterministico) è costituito da:

- Nastro in sola lettura in un'unica direzione
- Memoria in scrittura a pila (LIFO)

L'automa è formato dalla quintupla della NFA a cui vengono aggiunti un alfabeto per la pila e un simbolo iniziale per la pila.

$$\text{APND} = \langle Q, \Sigma, R, \delta, q_0, F, z_0 \rangle$$

- Q : è l'insieme **finito** degli stati
- Σ : è l'insieme **finito** di simboli (alfabeto)
- R : è l'insieme **finito** di simboli della pila
- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times R \rightarrow \mathcal{P}(Q \times R^*)$: è la funzione di transizione. Prende in input lo stato in cui si trova la macchina (Q), l'eventuale simbolo sul nastro ($\Sigma \cup \{\epsilon\}$) e il simbolo in cima alla pila (R) e restituisce un insieme di coppie ($Q \times R^*$) che rappresentano il nuovo stato e il push di una sequenza di simboli nella pila
- $q_0 \in Q$: è lo stato iniziale
- $F \subseteq Q$: è l'insieme degli stati finali
- $z_0 \in R$: è il simbolo iniziale della pila

La chiusura transitiva di δ fornisce un'esecuzione dell'automa.

Definizione 3.7 (Linguaggio riconosciuto da un APND). Una stringa si può riconoscere quando l'automa arriva in uno stato finale oppure quando la pila è vuota. Consideriamo un APND M , allora si hanno due modi per definire il linguaggio riconosciuto da M :

- $L_p(M)$ linguaggio per pila vuota

$$L_p(M) = \{\sigma \in \Sigma^* \mid (q_0, \sigma, z_0) \xrightarrow{*} (q, \epsilon, \epsilon), q \in Q\}$$

cioè l'insieme delle stringhe σ tali che partendo dallo stato iniziale q_0 , con la stringa σ sul nastro e il simbolo iniziale z_0 sulla pila, si arriva in uno stato qualsiasi $q \in Q$ con la pila vuota e il nastro vuoto.

- $L_f(M)$ linguaggio per stato finale

$$L_f(M) = \{\sigma \in \Sigma^* \mid (q_0, \sigma, z_0) \xrightarrow{*} (q, \epsilon, \gamma), q \in F, \gamma \in R^*\}$$

La scelta della modalità di riconoscimento non cambia la potenza espressiva:

$$\forall M \text{ APND} . \exists M' \text{ APND} . L_p(M) = L_f(M')$$

Esempio 3.15. Consideriamo il linguaggio:

$$L = \{\sigma c \sigma^R \mid \sigma \in \{a, b\}^*\}$$

σ^R è definito come il *reverse* di σ , cioè la stringa letta al contrario. Quindi questo linguaggio contiene tutte le stringhe palindrome, ad esempio:

$$\sigma = abb \implies \sigma^R = bba \implies abbcbba \in L$$

La grammatica che genera questo linguaggio è:

$$S \rightarrow aSa \mid bSb \mid c$$

L'esempio di prima costruito con la grammatica è:

$$S \rightarrow aSa \rightarrow abSba \rightarrow abbSbba \rightarrow abbcbba$$

Definiamo l'automa a pila **deterministico** che riconosce questo linguaggio:

$$M = \langle \{q_0, q_1\}, \{a, b, c\}, \{Z, A, B\}, q_0, Z, \emptyset, \delta \rangle$$

che si rappresenta con la seguente tabella (una per ogni stato):

q_0	a	b	c
Z	q_0, ZA	q_0, ZB	q_1, ϵ
A			
B			

q_1	a	b	c
Z	q_1, Z	q_1, Z	
A	q_1, ϵ	q_1, Z	
B	q_1, Z	q_1, ϵ	

Questo automa riconosce il linguaggio quando la pila è vuota.

Esempio 3.16. Consideriamo il linguaggio:

$$L = \{\sigma\sigma^R \mid \sigma^R \text{ è il reverse di } \sigma, \sigma \in \{a, b\}^*\}$$

L'automa a pila **non deterministico** che riconosce questo linguaggio è:

$$M = \langle \{q_0, q_1\}, \{a, b\}, \{Z, A, B\}, q_0, Z, \emptyset, \delta \rangle$$

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_1	ϵ	a	b
Z	q_1, ϵ		
A		q_1, ϵ	
B			q_1, ϵ

Questo automa riconosce il linguaggio quando viene letto dal nastro ϵ e nella pila rimane solo ϵ .

Gli automi a pila deterministici sono sottoinsiemi degli automi a pila non deterministici:

$$\underbrace{\text{APD}}_{\text{Parser}} \subset \underbrace{\text{APND}}_{\text{CF}}$$

I linguaggi riconosciuti dagli APD sono i linguaggi di programmazione.

3.12 Corrispondenza tra APND e grammatiche context free

Teorema 3.8. Consideriamo un linguaggio riconosciuto da un APND M :

$$L = L(M)$$

allora esiste M' APND **con un solo stato** tale che:

$$L = L(M')$$

Dimostrazione: Consideriamo l'APND $M = \langle Q, \Sigma, R, \delta, q_0, z_0, F \rangle$. con:

- $|Q| \geq 1$
- $\delta : Q \times (\Sigma \times \{\varepsilon\}) \times R \rightarrow \mathcal{P}(Q \times R^*)$

L'idea è quella di prendere un nuovo simbolo per la pila che rappresenta ogni coppia $Q \times R$:

$$\begin{pmatrix} q_i, z_i \\ \in Q, \in R \end{pmatrix} \rightsquigarrow \text{Aggiungiamo } z'_i \\ R' = R \cup \{z'_i\}$$

Bisogna modificare la funzione di transizione δ in modo da rappresentare una coppia (q_i, z_i) con il nuovo simbolo z'_i associato ad un unico stato.

$$\delta(q_i, a, z_i) = \delta'(q_0, a, z'_i)$$

Essenzialmente si codifica nella pila la coppia stato-simbolo.

Teorema 3.9. Se $L = L(M)$ con M APND con uno stato, allora L è context free. Quindi esiste una grammatica context free G che riconosce L :

$$\exists G \text{ CF} . L = L(G)$$

Dimostrazione: Consideriamo l'APND $M = \langle \{q_0\}, \Sigma, R, \delta, q_0, z_0, \emptyset \rangle$ (accettiamo L per pila vuota) che riconosce L :

$$L = L(M)$$

La grammatica è definita come

$$G = \langle V, \Sigma, P, S \rangle \equiv \langle R, \Sigma, P, z_0 \rangle$$

con P :

$$\textcolor{red}{z} \rightarrow \textcolor{blue}{a} z_1 z_2 \dots z_n \in P \iff (q_0, z_1 z_2 \dots z_n) \in \delta(q_0, \textcolor{blue}{a}, \textcolor{red}{z})$$

Se si legge a sul nastro e z in cima alla pila, allora si può fare il push di $z_1 z_2 \dots z_n$ nella pila.

Teorema 3.10. Se L è context free allora esiste una APND M che riconosce il linguaggio $L = L(M)$, cioè dall'automa si può costruire una grammatica.

Dimostrazione: Consideriamo il linguaggio context free L generato dalla grammatica in forma normale di Greibach $G = \langle V, T, P, S \rangle$. $L = L(G)$. Costruiamo M :

- $Q = \{q\}$
- $\Sigma = T$
- $R = V$
- $z_0 = S$
- $F = \emptyset$

Dove ogni stato è:

$$\left(q, \underset{\in R^* = V^*}{\alpha} \right) \in \delta \left(q, \underset{\in T = \Sigma}{a}, \underset{\in R = V}{A} \right) \iff A \xrightarrow[\in \Sigma \in V^*]{} \alpha \in P$$

Esempio 3.17.

$$A \xrightarrow{} a\alpha \mid a\beta \rightsquigarrow \begin{cases} (q, \alpha) \in \delta(q, a, A) \\ (q, \beta) \in \delta(q, a, A) \end{cases}$$

4 Teoria della calcolabilità

4.1 Potenza espressiva

Esistono diversi modelli di calcolo che hanno diversa potenza espressiva, ad esempio

- Automi a stati finiti deterministici
- Grammatiche context free
- Automi a pila deterministici
- ecc...

Ogni modello viene istanziato per riconoscere uno specifico linguaggio:

- $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ per gli automi a stati finiti
- $G = \langle V, T, P, S \rangle$ per le grammatiche context free

- $M = \langle Q, \Sigma, R, \delta, q_0, z_0, F \rangle$ per gli automi a pila
- ecc...

Il risultato di queste istanze è una **funzione calcolata** (che era superflua per i modelli visti fin'ora):

Modello \rightsquigarrow Istanza del modello \rightsquigarrow Funzione calcolata

Per ragionare su ciò che sta fuori dai modelli visti, la funzione calcolabile diventa rilevante. In questo modo "trasformiamo" il calcolo di f nel riconoscimento di un linguaggio L_f

$$L_f = \{a^f(n) \mid n \in \mathbb{N}\}$$

I ragionamenti fatti fin'ora erano:

- Quali funzioni corrispondono a L_f regolari
- Quali funzioni corrispondono a L_f context free

$$f(n) = 2n \implies L_f = \{a^{2n} \mid n \in \mathbb{N}\} \text{ regolare}$$

$$f(n) = n^2 \implies L_f = \{a^{n^2} \mid n \in \mathbb{N}\} \text{ non CF}$$

Ora invece consideriamo un contesto più generale parlando di **insiemi** (di \mathbb{N}) invece che di linguaggi. Quindi il calcolo diventa:

$$A_f = \{f(n) \mid n \in \mathbb{N}\}$$

Si abbandona il concetto di stringa e ci si focalizza sul calcolo di una funzione f .

4.2 Teoria della Calcolabilità

- **Funzione calcolata:** è un'associazione input-output:

$$f : \underbrace{\mathbb{N}}_{\text{Dominio}} \rightarrow \underbrace{\mathbb{N}}_{\text{Codominio}} \quad f \subseteq \mathbb{N} \times \mathbb{N}$$

Questa associazione è definibile da tutte le coppie di input e output.

- **Algoritmo (processo di calcolo):** Alcune funzioni sono definibili con una sequenza di passi partendo dall'input, ad esempio:

$$\begin{cases} f(0) = 1 \\ f(1) = 1 \\ f(x+2) = f(x+1) + f(x) \text{ per } x \geq 0 \end{cases}$$

Si introduce quindi il concetto di **calcolabilità**. Per definire la calcolabilità ci domandiamo quali funzioni $f : \mathbb{N} \rightarrow \mathbb{N}$ sono calcolabili (mediante un algoritmo possiamo costruire l'output a partire dall'input)

Ci sono due modelli principali per definire la calcolabilità:

- **Funzioni primitive ricorsive**
- **Macchina di Turing**

4.3 Funzioni primitive ricorsive

Le funzioni primitive ricorsive sono un modello di calcolo che permette di descrivere un processo di calcolo componendo funzioni già definite o funzioni di base. Per rappresentare la composizione di funzioni si usa la **Lambda notazione**:

$$\underbrace{\lambda}_{\text{Funzione}} \underbrace{x}_{\text{Input}} . \underbrace{f(x)}_{\text{Output}}$$

4.3.1 Funzioni primitive di base

Le funzioni ricorsive di base sono:

- **Funzione costante zero:**

$$\lambda x.0$$

- **Funzione successore:**

$$\lambda x.x + 1 \equiv S(x)$$

- **Funzione identità (su un valore) o funzione proiezione (su più valori):**

$$\lambda x_1, x_2, \dots, x_n. x_i \quad 1 \leq i \leq n$$

O per un solo valore:

$$\lambda x.x$$

Queste funzioni sono **totali**, cioè sono definite per ogni valore naturale in input.

4.3.2 Operazioni sulle funzioni

Le possibili operazioni per costruire nuove funzioni sono:

- **Composizione:** Consideriamo le funzioni:

$$g : \mathbb{N} \rightarrow \mathbb{N} \quad h : \mathbb{N} \rightarrow \mathbb{N}$$

Abbiamo la funzione composta:

$$f = h \circ g = \lambda x. g(h(x)) \quad f : \mathbb{N} \rightarrow \mathbb{N}$$

In generale:

$$g : \mathbb{N}^k \rightarrow \mathbb{N} \quad h_1, h_2, \dots, h_k : \mathbb{N}^n \rightarrow \mathbb{N}$$

La funzione composta è:

$$\begin{aligned} f &= g \circ (h_1, \dots, h_k) \\ &= \lambda x_1, \dots, x_n. g(h_1(x_1, \dots, x_n), h_2(x_1, \dots, x_n), \dots, h_k(x_1, \dots, x_n)) \end{aligned}$$

La composizione di funzioni totali è totale.

Esempio 4.1. Consideriamo le funzioni:

$$g : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \quad h_1, h_2 : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$

La funzione composta è:

$$f = g \circ (h_1, h_2) = \lambda x_1, x_2, x_3. g(h_1(x_1, x_2, x_3), h_2(x_1, x_2, x_3))$$

$$f : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$

Queste funzioni equivalgono agli assegnamenti nei linguaggi di programmazione.

- **Primitiva ricorsione:** Consideriamo le funzioni:

$$g : \mathbb{N} \rightarrow \mathbb{N} \quad h : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$

Definiamo la funzione f mediante primitiva ricorsione come:

$$\begin{cases} f(\underbrace{0}_{\text{indice}}, x) = g(x) & \text{Caso base} \\ f(y + 1, x) = h(y, f(y, x), \underbrace{x}_{\text{input}}) & \text{Caso ricorsivo} \end{cases}$$

Le funzioni definite per primitiva ricorsione sono totali.

Queste funzioni equivalgono al ciclo for nei linguaggi di programmazione.

Definizione 4.1. Una funzione primitiva ricorsiva $f \in PR$, dove PR è la classe delle funzioni primitive ricorsive, ovvero la minima classe di funzioni chiuse per composizione e primitiva ricorsione che contiene tutte le funzioni di base.

Definizione 4.2. Supponiamo che una funzione f sia primitiva ricorsiva, allora esiste una sequenza finita di funzioni P_f in PR tali che l'ultima funzione della sequenza è esattamente f :

$$\exists P_f = f_1, f_2, \dots, f_n . \forall i \in PR \wedge f_n = f$$

Definizione utile 4.1. Quando una funzione totale termina si usa la seguente notazione:

$$f(x) \downarrow$$

Quando una funzione non è definita e quindi non termina si usa la seguente notazione:

$$f(x) \uparrow$$

4.3.3 Esempi

Esempio 4.2 (Somma). Definiamo la funzione somma:

$$+ : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$

Definiamola prima induttivamente:

$$\begin{cases} +(0, x) &= x \\ +(y + 1, x) &= S(+((y, x))) \end{cases}$$

Definiamo ora la funzione mediante funzioni ricorsive di base:

$$\begin{aligned} f_1 &= \lambda x. x \\ f_2 &= \lambda x. x + 1 \\ f_3 &= \lambda x_1 x_2 x_3. x_2 \\ f_4 &= f_2 \circ f_3 \end{aligned}$$

$$+ = f_5 : \begin{cases} f_5(0, x) = f_1(x) = x \\ f_5(y + 1, x) = f_4(y, f_5(y, x), x) \\ \quad = f_2 \circ f_3(y, f_5(y, x), x) \\ \quad = f_2(f_5(y, x)) = S(+((y, x))) \end{cases}$$

quindi è definita come:

$$+ : f_1, f_2, f_3, f_4, f_5$$

Esempio 4.3 (Moltiplicazione). Definiamo la funzione moltiplicazione:

$$* : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$

Definiamola prima induttivamente:

$$\begin{cases} *(0, x) &= 0 \\ *(y + 1, x) &= +((x, *(y, x))) \end{cases}$$

Definiamo ora la funzione mediante funzioni ricorsive di base:

$$\begin{aligned} f_1 &= \lambda x. 0 \\ f_2 &= \lambda x_1 x_2 x_3. x_2 \\ f_3 &= \lambda x_1 x_2 x_3. x_3 \\ f_4 &= + = \lambda x_1 x_2. + (x_1, x_2) \\ f_5 &= f_4 \circ (f_2, f_3) \end{aligned}$$

$$* = f_6 : \begin{cases} f_5(0, x) = f_1(x) = 0 \\ f_6(y + 1, x) = f_5(y, f_6(yx), x) \\ \quad = f_4 \circ (f_2, f_3)(y, f_6(yx), x) \\ \quad = f_4(f_2(y, f_6(yx), x), f_3(y, f_6(yx), x)) \\ \quad = f_4(f_6(yx), x) = +(f_6(yx), x) \end{cases}$$

quindi è definita come:

$$*: f_1, f_2, f_3, f_4, f_5, f_6$$

Esempio 4.4 (Decreimento). Definiamo la funzione decremento:

$$D : \mathbb{N} \rightarrow \mathbb{N}$$

Definiamola prima induttivamente:

$$\begin{cases} D(0) &= 0 \\ D(y + 1) &= y \end{cases}$$

Definiamo ora la funzione mediante funzioni ricorsive di base:

$$\begin{aligned} f_1 &= \lambda x.0 \\ f_2 &= \lambda x_1 x_2 x_3. x_1 \end{aligned}$$

$$D = f_3 : \begin{cases} f_3(0, x) = f_1(x) = 0 \\ f_3(y + 1, x) = f_2(y, f_3(y, x), x) = y \end{cases}$$

quindi è definita come:

$$D : f_1, f_2, f_3$$

Esercizio 4.1 (Differenza). Definiamo la funzione differenza:

$$- : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$

Definiamola prima induttivamente:

$$\begin{cases} -(0, x) &= x \\ -(x, y + 1) &= D(-(y, x)) \end{cases}$$

4.3.4 Calcolabilità delle funzioni primitive ricorsive

Vogliamo mostrare che questo modello fallisce nel catturare il concetto di calcolabilità. Consideriamo una funzione $f : \mathbb{N} \rightarrow \mathbb{N}$ primitiva ricorsiva $f \in PR$, sappiamo che:

$$\exists P_f = f_1, f_2, \dots, f_n . \forall i \in PR \wedge f_n = f$$

Definiamo una nuova funzione $h : \mathbb{N} \rightarrow \mathbb{N}$ tale che:

$$h(n) = f_n(n) + 1$$

Prendiamo un indice n , generiamo la lista di funzioni fino a n e gli sommiamo 1. Di conseguenza $h(n)$ è calcolabile e visto che funzioni primitive ricorsive catturano tutto ciò che è calcolabile, allora $h \in PR$. Quindi vuol dire che esiste un indice i tale che:

$$\exists i . h = f_i \in PR$$

Allora possiamo calcolare $h(i)$:

$$f_i(i) = h(i) = f_i(i) + 1 \quad \text{Assurdo}$$

È impossibile che un numero naturale sia uguale al suo successore. Quindi PR non può catturare tutte le funzioni calcolabili.

Esempio 4.5. Un esempio di funzione totale non primitiva ricorsiva è la funzione di Ackermann:

$$Ack(0, y) = y + 1 \quad \text{Caso base per il primo parametro}$$

$$Ack(x + 1, 0) = Ack(x, 1) \quad \text{Caso base per il secondo parametro}$$

$$Ack(x + 1, y + 1) = Ack(x, Ack(x + 1, y)) \quad \text{Caso ricorsivo}$$

Questa funzione cresce molto velocemente, ad esempio:

$$Ack(4, 2) \text{ ha più di 19000 cifre}$$

Nella definizione, al posto di h , abbiamo una funzione da calcolare e quindi rappresenta la f della definizione di primitiva ricorsione, cioè la funzione che calcolo è quella calcolata. Quindi lo schema di primitiva ricorsione non è rispettato:

$$Ack \notin PR$$

però Ack è totale, calcolabile, ma non primitiva ricorsiva.

4.4 Macchina di Turing

È un modello basato su automi a stati finiti (DFA) che permette di descrivere come avviene il processo di calcolo come sequenza di operazioni di scrittura e lettura su una memoria. La macchina di Turing è costituita da:

- **Programma finito:** rappresenta la funzione di transizione, cioè una tabella finita in cui viene definito come output:
 - Stato: è una testina
 - Simbolo (scritto)
 - Direzione di spostamento
- **Nastro infinito:** rappresenta la memoria della macchina, è costituito da celle che contengono simboli di un alfabeto finito. La testina può leggere e scrivere su questo nastro.

Definizione 4.3 (Macchina di Turing). Una macchina di Turing (MdT) M consiste di:

- Σ : alfabeto **finito**:

$$\Sigma = \{s_0, s_1, \dots, s_k\}$$

L'alfabeto contiene almeno due simboli:

- $\$$: rappresenta la cella vuota del nastro
- 0 : simbolo significativo

- Q : insieme finito di stati:

$$q_0 \in Q \quad Q \neq \emptyset$$

- P : insieme finito di istruzioni (corrisponde alla δ del DFA):

$$P = \{I_1, \dots, I_k\}$$

Ogni istruzione è una quintupla che può essere di due tipi:

- $q, s, q', s', \{R, L\}$: dove:

- * q : stato corrente
- * s : simbolo letto sul nastro
- * q' : stato successivo
- * s' : simbolo da scrivere sul nastro sovrascrivendo s
- * R o L : direzione di spostamento della testina (Right o Left)

- δ : la funzione di transizione è definita come:

$$\delta : Q \times \Sigma \rightarrow Q \times \Sigma \times \{R, L\}$$

4.4.1 Descrizione istantanea di una MdT

La descrizione istantanea (ID) di una MdT rappresenta lo stato corrente della macchina in un dato istante di tempo:

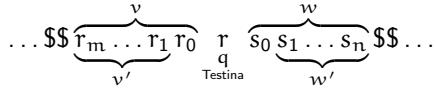
$$\dots \$\$ \underbrace{s_1 s_2 \dots s_{i-1}}_v \underbrace{s_i}_q \underbrace{s_{i+1} \dots s_n}_w \$\$ \dots$$

La descrizione istantanea contiene:

$$(q, v, s_i, w)$$

- Lo stato corrente q
- Il contenuto del nastro a sinistra della testina
- Il simbolo sotto la testina s_i
- Il contenuto del nastro a destra della testina

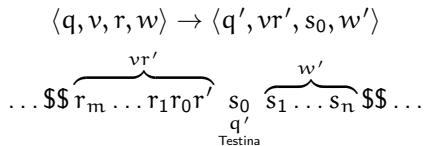
La computazione di una MdT è una sequenza di descrizioni istantanee.



- Per lo spostamento a destra:

$$\delta(q, r) = \langle q', r', R \rangle$$

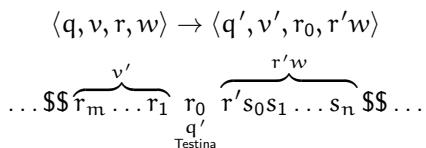
quindi:



- Per lo spostamento a sinistra:

$$\delta(q, r) = \langle q', r', L \rangle$$

quindi:



Definizione 4.4. Una computazione è terminante se esiste una descrizione istantanea $\langle q, v, r, w \rangle$ tale che nessuna istruzione inizia con (q, r) , ovvero $\delta(q, r)$ non è definita.

Teorema 4.1 (Funzione Turing calcolabile). Una funzione $f : \mathbb{N}^n \rightarrow \mathbb{N}$ è **Turing calcolabile** se esiste una MdT M tale che partendo dalla configurazione iniziale:

$$\$x_1\$x_2\$ \dots \$x_n\$ \$ \dots$$

\uparrow
 q_0

dove $x \in \mathbb{N}$ è la rappresentazione unaria, ad esempio:

$$\begin{aligned} |x| = 2 &\rightarrow x = 11 \\ |x| = 5 &\rightarrow x = 11111 \end{aligned}$$

e quindi x_1, \dots, x_n sono gli input di f , allora la MdT termina nella configurazione:

$$\dots \$f(x_1, \dots, x_n)\$ \$ \dots$$

\uparrow
 q_f

Una MdT termina quando nessuna computazione ulteriore è possibile.

Le funzioni primitive ricorsive possono essere rappresentate con una macchina di Turing.

Esempio 4.6. La funzione:

$$\lambda x.0$$

è rappresentata dalla macchina:

	\$
q ₁	q ₁ \$, R

Il nastro è:

... \$x \$ \$...
 ↑
 q₀
... \$x \$ \$...
 Risultato ↑
 q₁

Esempio 4.7. La funzione:

$$\lambda x.x$$

è rappresentata dalla macchina:

	\$
q ₁	

Il nastro è:

... \$x \$...
 ↑
 q₁

Esempio 4.8. Un esempio di MdT che non termina è:

	\$
q ₀	q ₀ \$, R

Scritta sotto forma di funzione:

$$f(x) = \lambda x. \uparrow$$

o in un linguaggio di programmazione:

1 while true do

```

2      x := x
3

```

4.5 Funzioni parziali ricorsive

Le funzioni parziali ricorsive sono funzioni primitive ricorsive con l'aggiunta dell'operazione di **minimizzazione**.

Definizione 4.5 (Minimizzazione). Sia $f : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$ una funzione totale^a, definiamo una funzione parziale $\varphi : \mathbb{N}^n \rightarrow \mathbb{N}$ o diverge:

$$\varphi(x_1 \dots x_n) = \mu z. (f(x_1 \dots x_n, z) = 0)$$

dove μz rappresenta il più piccolo z (se esiste) tale che $f(x_1 \dots x_n, z) = 0$. Se non esiste tale z allora $\varphi(x_1 \dots x_n) \uparrow$.

L'algoritmo che calcola φ è:

```

1 input(x_1, ..., x_n)
2   z = 0
3   while f(x_1, ..., x_n, z) != 0 do
4     z++
5
6   output(z)

```

^aDi solito le lettere f, g, h indicano funzioni totali, mentre le lettere greche, di solito φ, ψ , indicano funzioni parziali.

Esempio 4.9. Un esempio di funzione calcolata con la minimizzazione è:

$$\lfloor \log_a x \rfloor$$

cioè l'intero inferiore del logaritmo in base a di x (l'esponente più piccolo che non fa superare il valore di x).

Definiamo la funzione con la minimizzazione:

$$f(x) = \mu z. \text{lte}(x, a^{z+1}) = 0$$

dove lte è la funzione minore o uguale:

$$\text{lte}(x, y) = \begin{cases} 0 & \text{se } x \leq y \\ 1 & \text{se } x > y \end{cases}$$

Esempio 4.10. Un altro esempio è la funzione radice:

$$\lfloor \sqrt[n]{x} \rfloor$$

sappiamo che:

$$x = a^n \iff \sqrt[n]{x} = a$$

Definiamo la funzione con la minimizzazione:

$$f(x) = \mu z. (\text{lte}(x, (z+1)^n) = 0)$$

Teorema 4.2. Una funzione $f : \mathbb{N} \rightarrow \mathbb{N}$ è una **parziale ricorsiva** se e solo se f è Turing calcolabile.

Dimostrazione:

- $\implies (f \text{ parziale ricorsiva} \Rightarrow \text{Turing calcolabile})$:

– **Casi base:**

* Funzioni di base: esistono MdT che le calcolano

– **Composizione:** Consideriamo le funzioni parziali ricorsive f e g . Per ipotesi induttiva sappiamo che esiste una macchina di turing M_f che calcola f e una macchina di turing M_g che calcola g . Vogliamo mostrare che la funzione composta $f \circ g$ è parziale ricorsiva

$$f \circ g \text{ è parziale ricorsiva} \implies \exists M_{f \circ g}$$

Definiamo la MdT che calcola $f \circ g$ in pseudocodice:

$$M_{f \circ g} = \begin{cases} \text{input}(x) \\ \text{temp} = M_g(x) \\ M_f(\text{temp}) \end{cases}$$

– **Primitiva ricorsione:** Consideriamo le funzioni parziali ricorsive g e h . Per ipotesi induttiva sappiamo che esiste una macchina di turing M_h che calcola h e una macchina di turing M_g che calcola g . Vogliamo mostrare che la funzione definita mediante primitiva ricorsione f è parziale ricorsiva

$$f(x, n) = \begin{cases} g(x) & n = 0 \\ h(x, n - 1, f(x, n - 1)) & n > 0 \end{cases}$$

La MdT che calcola f è:

$$M_f = \begin{cases} \text{input}(x) \\ \text{input}(n) \\ \text{temp} = M_g(x) \\ \text{for } i = 0 \text{ to } n \text{ do} \\ \quad \{\text{temp} := M_h(x, i, \text{temp})\} \\ \text{output}(\text{temp}) \end{cases}$$

- **Minimizzazione:** Consideriamo la funzione parziale ricorsiva f . Per ipotesi induttiva sappiamo che esiste una macchina di turing M_f che calcola f . Vogliamo mostrare che la funzione definita mediante minimizzazione φ è parziale ricorsiva

$$\varphi(x) = \mu z. (f(x_1 \dots x_k, z) = 0)$$

La MdT che calcola φ è:

$$M_\varphi = \begin{cases} \text{input}(x_1 \dots x_k) \\ i = 0 \\ \text{temp} = M_f(x_1 \dots x_k, i) \\ \text{while temp } \neq 0 \text{ do} \\ \quad i++ \quad \text{temp} = M_f(x_1 \dots x_k, i) \\ \} \\ \text{output(temp)} \end{cases}$$

- $\Leftarrow (f \text{ Turing calcolabile} \Rightarrow \text{parziale ricorsiva})$: Data la descrizione istantanea:

$$\langle q, n, s, m \rangle \rightarrow \langle q', n', s', m' \rangle \quad \text{primo passo}$$

dove:

- I simboli della macchina di Turing sono $\{\$, 0, 1\}$
- n, m le sequenze di simboli sul nastro sono numeri binari con il numero meno significativo vicino alla testina
- Codifichiamo gli spostamenti della testina come:
 - * $L \rightarrow 0$
 - * $R \rightarrow 1$

Scorpiamo la funzione le funzioni in $\delta : Q \times \Sigma \rightarrow Q \times \Sigma \times \{R, L\}$:

- $\mathbb{Q} : Q \times \Sigma \rightarrow Q$ funzione totale, quindi PR. Rappresenta la transizione di stato
- $\mathbb{S} : Q \times \Sigma \rightarrow \Sigma$ funzione totale, quindi PR. Rappresenta la transizione di simbolo
- $\mathbb{X} : Q \times \Sigma \rightarrow \{R, L\}$ funzione totale, quindi PR. Rappresenta transizione di direzione

Distinguiamo due casi:

1. Se $\mathbb{X}(q, s) = L = 0$ allora:

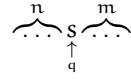
$$q' = \mathbb{Q}(q, s)$$

$$m' = 2m + \mathbb{S}(q, s) \text{ (aggiungo il simbolo meno significativo di } m)$$

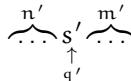
$$n' = n \text{div} 2 \text{ (} n \text{ senza il suo simbolo meno significativo)}$$

$$s' = n \text{mod} 2 \text{ (simbolo meno significativo di } n)$$

con il nastro:



Se ci si sposta a sinistra il nastro diventa:



Essendo le funzioni mod e div primitive ricorsive, allora tutte queste operazioni sono primitive ricorsive.

2. Se $\mathbb{X}(q, s) = R = 1$ allora:

$$q' = \mathbb{Q}(q, s)$$

$m' = \text{mdiv2}$ (m senza il suo simbolo meno significativo)

$n' = 2n + \mathbb{S}(q, s)$ (aggiungo il simbolo meno significativo di n)

$s' = \text{mmod2}$ (simbolo meno significativo di m)

Combiniamo questi due casi:

$$q' = \mathbb{Q}(q, s)$$

$$\begin{aligned} s' &= (n \bmod 2) \cdot \underbrace{(1 - \mathbb{X}(q, s))}_{\neq 0 \text{ se } L} + (m \bmod 2) \cdot \underbrace{\mathbb{X}(q, s)}_{\neq 0 \text{ se } R} \\ &= 0 \text{ se } R \\ &= 0 \text{ se } L \end{aligned}$$

$$m' = (2m + \mathbb{S}(q, s)) \cdot (1 - \mathbb{X}(q, s)) + (\text{mdiv2}) \cdot \mathbb{X}(q, s)$$

$$n' = (n \bmod 2) \cdot (1 - \mathbb{X}(q, s)) + (2n + \mathbb{S}(q, s)) \cdot \mathbb{X}(q, s)$$

quindi la funzione f definita come:

$$f : \langle q, n, s, m \rangle \rightarrow \langle q', n', s', m' \rangle$$

è primitiva ricorsiva.

Per primitiva ricorsione calcoliamo lo stato raggiunto dopo t passi:

$$P_q(t, q, n, s, m) = \begin{cases} P_q(0, q, n, s, m) = q \\ P_q(t+1, q, n, s, m) = P_q(t, \underbrace{q', n', s', m'}_{\text{Calcolati da } f}) \end{cases}$$

Quindi P_q esegue t passi della MdT a partire dalla descrizione istantanea $\langle q, n, s, m \rangle$ e restituisce lo stato raggiunto.

Supponendo l'output come ciò che sta a sinistra della testina quando la MdT termina, definiamo per primitiva ricorsione (di computazione parziale) dopo t passi:

$$P_{\text{out}}(t, q, n, s, m) =$$

$$= \begin{cases} P_{out}(0, q, n, s, m) = n \\ P_{out}(t + 1, q, n, s, m) = P_{out}(t, q', n', s', m') \end{cases}$$

Quindi P_{out} restituisce la sequenza a sinistra della testina dopo t passi di calcolo.

In Q supponiamo di avere uno stato q_f di terminazione. Quindi il calcolo di M è:

$$\varphi_M(x) = P_{out}(\mu t. (q_f - P_q(t, q_0, x \text{div} 2, x \text{mod} 2, 0)) = 0, \\ q_0, \\ x \text{div} 2, \\ x \text{mod} 2, \\ 0)$$

dove:

- $P_q(t, q_0, x, x \text{mod} 2, 0)$ cerca il più piccolo t tale che partendo da q_0 con input x a sinistra della testina in t passi arriva allo stato finale q_f
- $P_{out}(\dots)$ calcola l'output dopo t passi, i quali portano allo stato finale

Abbiamo quindi dimostrato che le macchine di Turing sono equivalenti alle funzioni parziali ricorsive:

$$MdT \equiv \text{Funzioni parziali ricorsive}$$

quindi se φ è una funzione parziale ricorsiva allora esiste una MdT che la calcola, cioè esiste un algoritmo che calcola φ . Se abbiamo una macchina di Turing M allora la funzione calcolata è parziale ricorsiva.

4.5.1 Aritmetizzazione delle macchine di Turing

Una macchina di Turing è costituita da due componenti principali:

- Programma, ad esempio Mdt , funzioni parziale ricorsive, programma in un qualsiasi linguaggio di programmazione
- Input, ad esempio un numero o una stringa, ecc...

Un programma può essere rappresentato nello stesso dominio degli input e quindi è possibile codificare (univocamente) ogni macchina di Turing in un numero naturale. Questo processo è chiamato **aritmetizzazione delle macchine di Turing**.

Esempio 4.11. Se consideriamo:

$$\sigma = \{0, \$\} \quad Q = \{q_0\}$$

Tutte le macchine di Turing possibili sono:

	0	\$
q ₀	•	•

dove • sono tutte le possibili istruzioni:

$$\begin{cases} (\$, \$, \$) \\ (q_0, \$, R) \\ (q_0, \$, L) \\ (q_0, 0, R) \\ (q_0, 0, L) \end{cases}$$

Il numero di macchine di Turing possibili da 1 stato e 2 simboli è:

$$5 \cdot 5 = 25$$

Possiamo mettere in ordine gli insiemi degli stati e dei simboli perchè sono finiti:

$$Q \rightarrow q_0 < q_1 < \dots < q_n \quad q_i \leq q_j \text{ se } i \leq j$$

$$\Sigma \rightarrow s_0 < s_1 < \dots < s_k \quad s_i \leq s_j \text{ se } i \leq j$$

Anche le triple possono essere messe in ordine lessicografico:

$$q_{i_1} s_{j_1} m_{k_1} \leq q_{i_2} s_{j_2} m_{k_2} \text{ se } \begin{cases} q_{i_1} \leq q_{i_2} \text{ oppure} \\ q_{i_1} = q_{i_2} \wedge s_{j_1} \leq s_{j_2} \text{ oppure} \\ q_{i_1} = q_{i_2} \wedge s_{j_1} = s_{j_2} \wedge m_{k_1} \leq m_{k_2} \end{cases}$$

4.5.2 Fattorizzazione di Gödel

Esistono diversi algoritmi per aritmetizzare una MdT, uno di questi è la **fattorizzazione di Gödel**. L'idea è quella di associare ad ogni macchina di Turing un numero naturale unico e codificare la macchina di Turing con gli esponenti dei numeri della fattorizzazione in numeri primi.

$$x \in \mathbb{N} \quad \text{fact}(x) = p_1^{x_1} \cdot p_2^{x_2} \cdot \dots \cdot p_n^{x_n} \quad p_i \text{ sono primi}$$

- **Fattorizzazione di Gödel di una istruzione:**

Associando ad ogni simbolo y della macchina di Turing un numero dispari maggiore di 1: $a(y)$. Un istruzione della macchina di Turing è una quintupla:

$$E = \langle q, s, q', s', R|L \rangle$$

Calcoliamo il numero associato all'istruzione E:

$$gn(E) = \prod_{i=1}^5 p_i^{a(s_i)}$$

Esempio 4.12. Consideriamo la seguente istruzione:

$$E = \langle q_0, \$, q_0, 0, R \rangle$$

Il numero di Gödel associato è:

$$gn(E) = 2^{a(q_0)} \cdot 3^{a(\$)} \cdot 5^{a(q_0)} \cdot 7^{a(0)} \cdot 11^{a(R)}$$

- **Fattorizzazione di Gödel di una macchina di Turing:**

Nello stesso modo si può calcolare il numero di Gödel associato ad una macchina di Turing M , cioè una sequenza di istruzioni:

$$M = E_1 E_2 \dots E_n \quad \text{Programma della MdT}$$

$$gn(M) = \prod_{i=1}^n p_i^{gn(E_i)}$$

- **Decodifica:**

Per decodificare un numero naturale x e ottenere la macchina di Turing associata, si esegue la fattorizzazione in numeri primi di x e si ricavano le istruzioni E_i dai fattori primi:

$$x = n_1 \cdot \dots \cdot n_k = \underbrace{m_1^1 \cdot \dots \cdot m_5^1}_{\text{Tupla 1}} \cdot \underbrace{m_1^2 \cdot \dots \cdot m_5^2}_{\text{Tupla 2}} \cdot \dots \cdot \underbrace{m_1^n \cdot \dots \cdot m_5^n}_{\text{Tupla n}}$$

Indichiamo con M_x la macchina di Turing il cui programma è codificato in $x \in \mathbb{N}$. M_x calcola la funzione φ_x , che è una notazione per indicare la funzione parziale ricorsiva calcolata dal programma codificato in $x \in \mathbb{N}$.

$$\varphi_x(y) = M_x(y) \quad (\text{Calcolo la MdT } M_x \text{ su input } y)$$

φ_x è la semantica (funzione calcolata) dal programma x .

Esempio 4.13. Se x_1 è il programma che calcola $y + y$ e x_2 fosse il programma che calcola $2y$ allora sappiamo che i programmi sono diversi:

$$x_1 \neq x_2$$

Però la funzione calcolata è la stessa:

$$\varphi_{x_1}(y) = \varphi_{x_2}(y)$$

4.5.3 Macchina universale (interprete)

La macchina universale è un modello di una macchina programmabile. È un programma che prende in input due valori, interpreta il primo come un programma e lo applica al secondo.

Definizione 4.6. Dati:

$$\begin{cases} x_1 & \text{Codifica } M_{x_1} \\ x_2 & \text{Input da manipolare} \end{cases}$$

Una macchina universale è definita come:

$$\varphi_u(x_1, x_2) = \varphi_{x_1}(x_2)$$

che sottoforma di pseudocodice diventa:

```

1 input(x_1) # Programma
2 input(x_2) # Dato
3
4 M = decodifica(x_1) # Algoritmo descritto da x_1
# costruisce il programma corrispondente
5 if ok(M) then M(x_2) # ok(M) e' il parser che verifica se M e'
# ben formata
6 else
7   while true
8     x = x # Diverge

```

4.6 Lambda calcolo

Il lambda calcolo è un linguaggio di programmazione funzionale. Si è dimostrato che il lambda calcolo è equivalente alle macchine di Turing. Esiste una tesi (teorema non dimostrabile, ma non ancora confutata) chiamata tesi di Church e dice che:

La classe delle funzioni "intuitivamente calcolabili" è equivalente alla classe delle funzioni Turing calcolabili.

4.7 Problemi insolubili

4.7.1 Problema della terminazione

Il problema della terminazione (Halting Problem) è il seguente: Non esiste una funzione totale ricorsiva e quindi calcolabile g tale che:

$$\forall x. g(x) = \begin{cases} 1 & \text{se } \varphi_x(x) \downarrow \text{ termina} \\ 0 & \text{se } \varphi_x(x) \uparrow \text{ diverge} \end{cases}$$

$\varphi_x(x) \downarrow$ rappresenta la funzione calcolabile della MdT di codice x e input x che termina. La funzione g decide (cioè risponde con 0 (diverge) o 1 (termina)) se φ_x applicata a x termina.

Dimostrazione per assurdo:

Supponiamo che esista una tale funzione g totale ricorsiva, allora esiste una macchina di Turing di codice $i_0 \in \mathbb{N}$ che la calcola.

$$g = \varphi_{i_0}$$

Possiamo definire una nuova funzione h :

$$h(x) = \begin{cases} 0 & \text{se } g(x) = 0 = \varphi_{i_0}(x) \\ \uparrow & \text{se } g(x) = 1 = \varphi_{i_0}(x) \end{cases}$$

Essendo g calcolabile, anche h è calcolabile, quindi:

$$\exists i_1. h = \varphi_{i_1}$$

Definiamo $h(x)$ in pseudocodice:

```

1 input(x)
2
3 if M_i0(x) == 1 then
4   output(0)
5 else
6   while true do
7     x = x # Diverge

```

L'ipotesi è che M_{i_0} esiste e termina sempre. Si ha un assurdo per costruzione. Supponiamo che $\varphi_{i_0}(i_0)$ converga:

$$\varphi_{i_1}(i_1) \downarrow \iff \underbrace{h(i_1) = 0}_{\text{per } h = \varphi_{i_1}} \iff \underbrace{g(i_1) = 0}_{\text{per def di } h} \iff \underbrace{\varphi_{i_1}(i_1) \uparrow}_{\text{per def di } g} \quad \text{Assurdo}$$

Supponiamo che $\varphi_{i_0}(i_0)$ diverga:

$$\varphi_{i_1}(i_1) \uparrow \iff \underbrace{h(i_1) \uparrow}_{\text{per } h = \varphi_{i_1}} \iff \underbrace{g(i_1) = 1}_{\text{per def di } h} \iff \underbrace{\varphi_{i_1}(i_1) \downarrow}_{\text{per def di } g} \quad \text{Assurdo}$$

Si dovrebbero aspettare infiniti passi per decidere se $\varphi_x(x)$ termina o diverge, quindi g non è calcolabile. Non esiste quindi un programma in grado di decidere **sempre** se un altro programma termina su un suo input:

$$\exists \psi. \psi(xy) = \begin{cases} 1 & \text{se } \varphi_x(y) \downarrow \\ 0 & \text{se } \varphi_x(y) \uparrow \end{cases}$$

Se esistesse $\psi(xy)$ allora potrei decidere se $\varphi_x(x)$ termina o diverge:

$$\psi(xx) = g(x)$$

e abbiamo dimostrato sopra che non è possibile.

Definizione 4.7. Non esiste f totale ricorsiva tale che:

$$\forall x. f(x) = \begin{cases} 1 & \text{se } \varphi_x(x) \text{ è totale} \\ 0 & \text{se } \varphi_x(x) \text{ non è totale} \end{cases}$$

$\varphi_x(x)$ è totale se:

$$\forall y. \varphi_x(y) \downarrow$$

e non è totale se:

$$\exists y. \varphi_x(y) \uparrow$$

Alcuni problemi **insolubili** sono:

- Decidere se f è costante, cioè se:

$$\exists n. \forall y. \varphi_x(y) = n$$

non è possibile.

- Decidere se φ_x non restituisce mai un valore fissato m , cioè se:

$$\forall y. \varphi_x(y) \neq m$$

- Decidere se $\varphi_x = \varphi_y$

4.8 Specializzatore

Teorema 4.3 (s-m-n (specializzatore)). Consideriamo una programma con due input:

$$f(x, y) = \lambda xy.x + y$$

Specializzare equivale a costruire un nuovo programma in cui una parte degli input è integrata nel codice. Usiamo f per ottenere f' che somma ad un valore in unput il valore 5.

$$f'(y) = \lambda y.5 + y$$

Si è quindi specializzata f a 5 per x . Se si specializza su tutti gli input, il programma diventa costante. La specializzazione su input parziali viene detta **partial computation**.

Quindi in modo formale: $\forall n, m \geq 1$ esiste una funzione **totale** ricorsiva s tale che

$$\begin{aligned} & \forall x, y_1, \dots, y_m . \lambda z_1 \dots z_n . \\ & \quad \underbrace{\varphi_x}_{\text{Codice}} \left(\underbrace{y_1 \dots y_m}_{\text{Input specializzati}} z_1 \dots z_n \right) \\ & = \lambda z_1 \dots z_n . \varphi_s(x, y_1 \dots y_m) \end{aligned}$$

Dove $\varphi_s(x, y_1 \dots y_m)$ è la trasformazione totale S del codice x e considerando gli input specializzati $y_1 \dots y_m$

Dimostrazione: Consideriamo $m = n = 1$ (caso particolare, ma $m > 1$ e $n > 1$ è analogo) con:

$$\lambda z. \varphi_x(y, z)$$

Fissiamo x_0 (programma) e y_0 (l'input su cui specializzare):

$$\implies \lambda z. \varphi_{x_0}(y_0, z)$$

dove φ_{x_0} denota la funzione calcolata dalla macchina di Turing il cui codice è aritmetizzato in x_0 . Questa funzione è **intuitivamente calcolabile**, di conseguenza esiste una macchina di Turing che la calcola (tesi di Church) il cui codice dipende (ed è costruibile ricorsivamente) da x_0 (codice originale) e y_0 (l'input specializzato). Quindi esiste una trasformazione **totale ricorsiva** tale che il codice è $s(x_0, y_0)$:

$$\varphi_{s(x_0, y_0)}(z) = \lambda z. \lambda z. \varphi_{x_0}(y_0, z)$$

4.9 Prima proiezione di Futamura

La proiezione di Futamura è una tecnica per ottenere un compilatore a partire da un interprete e uno specializzatore.

Definizione 4.8. Dato un programma sorgente $\text{source} \in L$ esiste una trasfor-

mazione di source in un programma (semanticamente equivalente) target $\in L'$.

Dimostrazione: Supponiamo che int sia la macchina universale:

$$\text{int}(p, d) = p(d)$$

dove:

- p è un programma
- d è un dato di input

Supponiamo che spec sia uno specializzatore:

$$\text{spec}(Q, d) = Q'_d \text{ integra nel codice l'input specializzato } d$$

dove:

- Q è un programma a due input

Allora:

$$\text{target} = \underbrace{\text{spec}(\text{int}, \text{source})}_{\text{Compilatore}}$$

4.9.1 Linguaggio Turing completo

Un linguaggio Turing completo permette di calcolare tutte le funzioni intuitivamente calcolabili. Un linguaggio Turing completo equivale ad una macchina di Turing. I costrutti minimi per avere un linguaggio Turing completo sono:

$$\begin{aligned} \exists p . \varphi_p &= \lambda x. 0 \\ \exists p . \varphi_p &= \lambda x. x + 1 \\ \exists p . \varphi_p &= \lambda x. x \text{ (proiezione)} \\ \exists p . \varphi_p &= \varphi_Q \circ \varphi_H \text{ (composizione)} \\ \exists p . \varphi_p &= \varphi_{\text{while}} B \{Q\} \text{ (minimizzazione)} \quad \exists p . \varphi_p(x, y) = \varphi_x(y) \\ \exists p . \varphi_{\text{spec}(x, y)}(z) &= \lambda z. \varphi_x(y, z) \text{ (proiezione)} \end{aligned}$$

Se consideriamo la **primitiva ricorsione** abbiamo solo cicli for. Un linguaggio che non permette cicli non determinati (while) **non** è Turing completo.

I linguaggi di Turing sono tutti quei linguaggi accettati da una mcchina di Turing:

$$\exists M dT m . L = L(M) = \{y \mid \varphi_m(y) \downarrow\}$$

dove y sono le stringhe su cui la MdT termina. E φ_x è la funzione parziale ricorsiva calcolata da una MdT di codice x .

Definizione 4.9. Il dominio di una funzione parziale ricorsiva φ_x è l'insieme:

$$\text{dom}(\varphi_x) = \{y \mid \varphi_x(y) \downarrow\}$$

5 Teoria della ricorsione

La teoria della ricorsione parla degli insiemi (linguaggi) caratterizzati dalle macchine di Turing.

5.1 Insiemi ricorsivamente enumerabili

Definizione 5.1. Un insieme $I \subseteq \mathbb{N}$ è ricorsivamente enumerabile (RE) se è il dominio di una funzione parziale ricorsiva, cioè:

$$\exists \varphi_x \text{ parziale ricorsiva . } \text{dom}(\varphi_x) = I$$

Cioè se esiste una macchina di Turing che si ferma su **tutti** gli elementi di I .

Notazione: Per rappresentare un insieme RE si usa la notazione:

$$W_x \stackrel{\text{def}}{=} \text{dom}(\varphi_x)$$

Se abbiamo φ_x parziale ricorsiva, allora W_x è RE:

$$\varphi_x \text{ PR} \implies W_x = \text{dom}(\varphi_x) \text{ è RE}$$

Se abbiamo $I \subseteq \mathbb{N}$ e riusciamo a costruire la sua **funzione semicaratteristica**, cioè:

$$\psi_I(x) = \begin{cases} 1 & \text{se } x \in I \\ \uparrow & \text{se } x \notin I \end{cases} \implies \text{dom}(\psi_I) = I$$

Per dimostrare che un insieme è RE bisogna fornire l'algoritmo della funzione semicaratteristica che deve terminare se e solo se ha in input un elemento dell'insieme.

Esempio 5.1. Consideriamo una variante del problema della terminazione, cioè quello che determina i programmi che terminano su **almeno un input**. Si può riscrivere con il seguente insieme:

$$\{x \mid \exists y . \varphi_x(y) \downarrow\}$$

Consideriamo il caso specifico applicato all'input 0:

$$I = \{x \mid \varphi_x(0) \downarrow\}$$

L'algoritmo della funzione semicaratteristica è:

¹ `input(x)`

```

2 stop = 0
3 build phi_x # Esiste algoritmo ricorsivo che costruisce la MdT
   di codice x
4 while !stop do {
5   execute next step of phi_x(0)
6   if phi_x(0) halts then {
7     output(1)
8     stop = 1
9   }
10 }
11 output(1)

```

La funzione è semicaratteristico perchè se $\varphi_x(0) \downarrow$ restituisce 1, altrimenti diverge.

Esempio 5.2. Consideriamo l'insieme del problema della terminazione generale:

$$I = \{x \mid \exists y . \varphi_x(y) \downarrow\}$$

Un possibile algoritmo è:

```

1 input(x)
2 y = 0
3 stop = 0
4 while !stop do {
5   execute next step of phi_x(y)
6   if phi_x(y) halts then {
7     stop = 1
8   }
9 }
10 return(1)

```

Questo programma è sbagliato perchè se diverge su 0, non potrà mai testare $y > 0$, ma $\varphi_x(1)$ potrebbe terminare. Quindi questo algoritmo diverge anche per $x \in I$ e di conseguenza non è la funzione semicaratteristica di I .

Un modo per risolvere questo problema è eseguire in parallelo tutte le $\varphi_x(y)$ per $y = 0, 1, 2, \dots$. Siccome non si può eseguire in parallelo, si può simulare l'esecuzione in parallelo eseguendo un passo di ogni $\varphi_x(y)$ in modo ciclico facendo partire una nuova computazione ad ogni ciclo. In questo modo se c'è almeno un input su cui prima o poi la macchina termina, prima o poi lo trova:

- **Passo 0:** Esegue un passo di $\varphi_x(0)$.
 - Se termina (\downarrow) allora finisce
 - Se non termina (\uparrow) allora si esegue il passo successivo
- **Passo n+1:** Esegue un passo per $\varphi_x(0), \varphi_x(1), \dots, \varphi_x(n)$ (se non sono terminate prima). Esegue il primo passo di $\varphi_x(n+1)$.

Questo algoritmo si chiama **Dovetail**.

La funzione semicaratteristica di I è:

```

1 input(x)
2 build phi_x # Esiste una procedura ricorsiva per costruire
    phi_x da x
3
4 y = 0 # Serve per identificare gli input su cui abbiamo
      iniziato la computazione
5 stop = 0
6
7 while !stop do {
8     for i from 0 to y {
9         execute next step of phi_x(i)
10        if phi_x(i) halts then {
11            stop = 1
12        }
13        y++
14    }
15 }
16 return(1)

```

5.2 Insiemi ricorsivi

Esiste un algoritmo che decide se un elemento sta o meno nell'insieme. L'insieme ricorsivo è caratterizzato da una funzione totale (\Rightarrow dominio in \mathbb{N}).

Definizione 5.2. Un insieme I è ricorsivo se esiste una funzione totale ricorsiva f_I tale che:

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

Questa funzione è detta **funzione caratteristica** di I .

Per dimostrare che un insieme è ricorsivo bisogna fornire l'algoritmo che calcola la sua funzione caratteristica.

Esempio 5.3. Un esempio di insieme ricorsivo è l'insieme dei numeri pari:

$$I = \{x \in \mathbb{N} \mid x \text{ è pari}\}$$

L'algoritmo che calcola la funzione caratteristica f è:

```

1 input(x)
2 y = x mod 2 # Resto della divisione per 2 (funzione PR)
3 z = neg(y) # y = 0 -> z = 1; y = 1 -> z = 0
4 output(z)

```

f è la caratteristica di I :

$$f(x) = \begin{cases} 1 & \text{se } x \in I \text{ (x è pari)} \\ 0 & \text{se } x \notin I \text{ (x è dispari)} \end{cases}$$

Esempio 5.4. Consideriamo il seguente insieme:

$$I = \{x \mid \exists n . x = n^2\}$$

Un esempio di algoritmo è:

```

1 input(x)
2 n = 0
3 while x != n^2 do n++
4 output(1)

```

Questo programma però non è la funzione caratteristica di I perché non può essere totale. Bisogna quindi trovare un modo per far terminare il programma anche se $x \notin I$.

```

1 input(x)
2 n = 0
3 while n <= x do {
4   if (x - n) then {
5     output(1)
6   }
7   n++
8 }
9 output(0)

```

Questo programma è la funzione caratteristica di I perché termina sempre:

$$f(x) = \begin{cases} 1 & \text{se } \exists n . x = n^2 \\ 0 & \text{altrimenti} \end{cases}$$

Teorema 5.1. Se l'insieme A è ricorsivo, allora è anche ricorsivamente enumerabile.

$$\exists f_A(x) = \begin{cases} 1 & \text{se } x \in A \\ 0 & \text{se } x \notin A \end{cases}$$

ma allora possiamo definire:

$$\psi_A(x) = \begin{cases} 1 & \text{se } f_A(x) = 1 \\ \text{while true } (\uparrow) & \text{se } f_A(x) = 0 \end{cases}$$

Supponiamo C_A è l'algoritmo che codifica f_A :

```

1 input(x)
2 y = C_A(x) # Esiste per ipotesi ed e' totale
3   if y = 1 then output(1)
4   else while true do x = x # Diverge

```

ed è la funzione semicaratteristica di A .

Teorema 5.2 (Teorema di Post). Un insieme A è ricorsivo se e solo se A e \bar{A} sono entrambi ricorsivamente enumerabili.

Dimostrazione: Se A è ricorsivo, allora A è RE (teorema precedente) e analogamente costruiamo la sua semicaratteristica per \bar{A} :

$$f_{\bar{A}}(x) = \begin{cases} 1 & \text{se } f_A(x) = 0 \\ \uparrow & \text{altrimenti} \end{cases}$$

Siano A e \bar{A} entrambi RE:

$$\begin{aligned} \psi_A(x) &= \begin{cases} 1 & \text{se } x \in A \\ \uparrow & \text{altrimenti} \end{cases} \\ \psi_{\bar{A}}(x) &= \begin{cases} 1 & \text{se } x \in \bar{A} (x \notin A) \\ \uparrow & \text{altrimenti} \end{cases} \end{aligned}$$

La funzione caratteristica di A è:

```

1 input(x)
2 stop = 0
3 while !stop do {
4   execute next step of C_A(x)
5   if C_A(x) halts then output(1)
6   execute next step of C_barA(x)
7   if C_barA(x) halts then output(0)
8 }
```

Esempio 5.5. Consideriamo l'insieme:

$$A_m = \{x \mid \varphi_x(x) \downarrow \text{ in meno di } m \text{ passi}\}$$

Questo insieme è ricorsivo perché m è un input che limita il numero di passi della computazione. Per dimostrarlo forniamo l'algoritmo della funzione caratteristica:

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

```

1 input(x, m)
2 build phi_x # Esiste procedura ricorsiva per costruire la MdT
  di codice x
3 for i from 1 to m do {
4   execute next step of phi_x(x)
5   if phi_x(x) halts then { # In questo caso la phi_x(x) ha
    terminato in meno di m passi
    output(1)
  }
}
output(0) # phi_x(x) e' arrivata a m passi senza terminare
```